с примерами на C++ и Emscripten

WebAssembly в действии

Жерар Галлан

MANNING

WebAssembly in Action

With examples using C++ and Emscripten

C. GERARD GALLANT

MANNING
SHELTER ISLAND

WebAssembly В ДЕЙСТВИИ

Жерар Галлан

с примерами на C++ и Emscripten

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.988.02-018
УДК 004.738.5
Г15

Галлан Жерар

Г15 WebAssembly в действии. — СПб.: Питер, 2022. — 496 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1691-1

Создавайте высокопроизводительные браузерные приложения, не полагаясь на один только JavaScript! Компилируясь в бинарный формат WebAssembly, ваш код на C, C++ или Rust будет работать в браузере с оптимальной скоростью. WebAssembly обеспечивает большую скорость, возможности повторного использования существующего кода и доступ к новым и более быстрым библиотекам. Кроме того, при необходимости вы можете настроить взаимодействие с JavaScript.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1617295744 англ.
ISBN 978-5-4461-1691-1

© 2019 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО Издательство «Питер», 2022
© Издание на русском языке, оформление ООО Издательство «Питер», 2022
© Серия «Библиотека программиста», 2022
© Павлов А., перевод с английского языка, 2021

Краткое содержание

Часть III. Продвинутые темы

Часть IV. Отладка и тестирование

Приложения

Оглавление

Часть II. Работа с модулями

Часть III. Продвинутые темы

Приложения

Предисловие

В сравнении с моими друзьями, я заинтересовался программированием относительно поздно. Я столкнулся с ним только в старших классах, поскольку мне был нужен еще один предмет, связанный с компьютерами, и мой куратор предложил информатику. Я думал, что нас будут обучать тому, как работает компьютер, но, к моему удивлению, это был курс по программированию. Вскоре я втянулся и в итоге сменил желаемую специальность с обычной архитектуры на архитектуру ПО.

В 2001 году я получил работу в Dovico Software по сопровождению и улучшению их клиент-серверного приложения на C++. Затем подул ветер перемен, в 2004 году в Dovico решили перевести его на модель «ПО как услуга» (SaaS), и я стал работать над этим веб-приложением. Я продолжал сопровождать приложения на C++, но моей основной сферой стала веб-разработка на C# и JavaScript. Сейчас я все еще занимаюсь веб-разработкой, но мой фокус сместился в сторону архитектуры — создания API, работы с базами данных и исследования новых технологий.

Я рад тому, что могу внести свой вклад в сообщество разработчиков с помощью блогов и лекций. В сентябре 2017 года меня спросили, не хочу ли я выступить перед локальной группой пользователей. В поиске тем для выступления я наткнулся на статью из PSPDFKit о технологии WebAssembly (https://pspdfkit.com/blog/2017/webassembly-a-new-hope/).

Я читал до этого о технологии Google Native Client (PNaCl), в которой скомпилированный код на C или C++ мог запускаться в браузере Chrome почти со скоростью нативного приложения. Я читал и о технологии Mozilla asm.js, позволяющей скомпилировать код на C или C++ в подмножество JavaScript, и он будет очень

быстро запускаться в браузерах, которые его поддерживают. В браузерах, не поддерживающих asm.js, код все равно будет запускаться, но с нормальной скоростью, поскольку это просто JavaScript. Тогда я и услышал о WebAssembly в первый раз.

Формат WebAssembly включает в себя все достижения asm.js и нацелен на устранение его недостатков. Он позволяет писать код на множестве различных языков и компилировать его в формат, который безопасно работает в браузере; кроме того, он уже доступен во всех основных браузерах, как «настольных», так и для мобильных устройств! Вдобавок он доступен и вне браузера, например на платформе Node.js! Меня просто потряс потенциал WebAssembly, и с тех пор я все свободное время проводил, пристально изучая эту технологию и описывая ее в блоге.

В конце 2017 года мои посты заметило издательство Manning Publications. Они связались со мной, чтобы узнать, будет ли мне интересно написать о WebAssembly книгу. Изначально в ней планировалось рассмотреть несколько языков, а также показать вам, как работать с этой технологией с точки зрения разработчика серверной и клиентской части приложения. Но после первой рецензии стала очевидна нехватка узконаправленности, поэтому мы решили, что будет лучше затронуть только язык программирования C++ и больше сосредоточиться на разработке серверной части.

Пока я работал над этой книгой, сообщество и рабочие группы WebAssembly не сидели сложа руки. Разрабатываются несколько обновлений. Например, стало возможным использовать многопоточные модули WebAssembly в «настольной» версии Google Chrome, не включая флаг экспериментальной функции! WebAssembly потенциально может помочь поднять веб-разработку на совершенно новый уровень, и я буду с интересом наблюдать за его дальнейшим развитием.

Благодарности

Мне говорили, что писать книгу тяжело и долго, но я не думал, что понадобится так много работать! Надеюсь, с помощью моих редакторов и рецензентов и благодаря отзывам людей, купивших первое издание, эта книга стала прекрасным пособием для тех, кто хочет начать работать с WebAssembly.

Я должен поблагодарить многих людей, благодаря которым создание этой книги стало возможным. В первую очередь моих родных — за то, что терпели меня, пока я работал над ней по вечерам, на выходных, и в праздники, и даже иногда в отпуске, чтобы успеть в срок. Моя жена Селена, дочери Донна и Одри, я вас очень люблю!

Затем хочу поблагодарить своего первого редактора в издательстве Manning, Кевина Харрельда, который помог мне собраться с силами и начать эту книгу. Кевин позже перешел в другую компанию, и я имел удовольствие работать над остатком книги с Тони Арритолой. Тони, спасибо тебе за терпение при работе со мной, за профессионализм, честность, когда ты не ходил вокруг да около и говорил все как есть, и за стремление получить качественный текст.

Благодарю всех в издательстве Manning, поучаствовавших в создании этой книги, от маркетинга до производства. Ваша неустанная работа очень важна.

Благодарю всех рецензентов, оторвавшихся на несколько часов от дел, чтобы прочитать данную книгу на разных этапах создания и дать конструктивную обратную связь, в том числе Кристофера Финка, Дэниела Баддена, Дарко Божиновски, Дэйва Катлера, Дениса Крейса, Германа Гонсалеса-Морриса, Джеймса Дитриха, Джеймса Хэринга, Джана Крокена, Джейсона Хейлса, Хавьера Муньоса, Джереми Лэнга, Джима Карабатсоса, Кэйт Майер, Марко Массенцио, Майка Рурка,

Милорада Имбру, Павло Ходиша, Питера Хэмптона, Резу Зейнали, Рональда Бормана, Сэма Зейдела, Сандера Зегвельда, Сатея Кумара Саху, Томаса Оверби Хансена, Тиклу Гангули, Тимоти Р. Кейна, Тишлиара Рональда, Кумара С. Унникришнана, Виктора Бека и Уэйна Мэтера.

Отдельная благодарность моему техническому редактору Иэну Ловеллу, чья обратная связь в процессе была бесценна, и техническому корректору Арно Бастенхофу, который перепроверял код перед отправкой книги в печать.

И наконец, огромная благодарность создателям браузеров, совместно работавших над созданием технологии, которая будет приносить пользу еще многие годы. Благодарю множество людей по всему миру, продолжающих работать над улучшением WebAssembly и расширением его функций. Возможности этой технологии огромны, и я с нетерпением жду, когда увижу, куда WebAssembly нас приведет.

О книге

Книга была написана, чтобы помочь вам понять, что такое WebAssembly, как он работает и что с ним можно и нельзя сделать. Она показывает разные варианты сборки модуля WebAssembly в зависимости от ваших потребностей. Мы начинаем с простых примеров и затем переходим к более сложным темам, например к динамическому связыванию, параллельной обработке и отладке.

ДЛЯ КОГО ЭТА КНИГА

Книга предназначена для разработчиков, имеющих базовое знание C и C++, JavaScript и HTML. По WebAssembly есть информация в Интернете, однако она частично устарела и обычно не очень подробна и не освещает сложные темы. В этой книге информация подана в удобочитаемом формате, который поможет как начинающим, так и опытным разработчикам создавать модули WebAssembly и взаимодействовать с ними.

СТРУКТУРА КНИГИ

В книге 13 глав, которые делятся на четыре части.

В части I объясняется, что такое WebAssembly и как он работает. Вдобавок в ней описан набор инструментальных средств Emscripten, с помощью которых вы будете на протяжении этой книги создавать модули WebAssembly.

- В главе 1 обсуждается, что такое WebAssembly, какие проблемы он решает и как работает. В ней также объясняется, что делает его безопасным, на каких языках можно создавать модули WebAssembly и где можно применять эти модули.

- В главе 2 объясняется, как структурирован модуль WebAssembly и за что отвечает каждый его раздел.
- В главе 3 вы познакомитесь с набором инструментальных средств Emscripten и узнаете о том, какие параметры вывода доступны при создании модуля WebAssembly. Вы также познакомитесь с API WebAssembly на JavaScript.

В части II показан процесс создания модуля WebAssembly и взаимодействия с ним в браузере.

- Из главы 4 вы узнаете, как взять существующую кодовую базу на C или C++ и изменить ее, чтобы она компилировалась в модуль WebAssembly. Кроме того, я покажу, как взаимодействовать с модулем из JavaScript вашей веб-страницы.
- В главе 5 объясняется, как изменить код, созданный в главе 4, чтобы модуль WebAssembly мог вызывать код на JavaScript с вашей веб-страницы.
- В главе 6 демонстрируется процесс модификации модуля WebAssembly, благодаря которому можно работать с указателями на функции, переданными модулю из вашего кода на JavaScript. Это позволит вашему JavaScript указывать функции по требованию и использовать промисы JavaScript.

В части III вы познакомитесь с более сложными темами, такими как динамическое связывание, параллельная обработка и работа с модулями WebAssembly за пределами браузера.

- Из главы 7 вы узнаете об основах динамического связывания, где два или более модуля WebAssembly могут быть связаны между собой в среде выполнения, чтобы действовать как один модуль.
- В главе 8 развивается тема, начатая в главе 7, и вы узнаете, как создавать несколько экземпляров одного и того же модуля WebAssembly и динамически связывать каждый экземпляр с другим модулем WebAssembly по требованию.
- Из главы 9 вы узнаете о веб-воркерах и pthreads. Вы научитесь предварительно загружать модули WebAssembly по мере надобности в фоновом потоке вашего браузера с помощью веб-воркеров. Вдобавок научитесь выполнять параллельную обработку в модуле WebAssembly, используя pthreads.
- В главе 10 показано, что WebAssembly не ограничен браузером. Вы научитесь использовать некоторые из ваших модулей WebAssembly в Node.js.

В части IV вы подробнее узнаете об отладке и тестировании.

- В главе 11 вы научитесь использовать текстовый формат WebAssembly, создав игру со сравнением карт.

- В главе 12 игра расширяется, чтобы показать вам различные параметры, доступные для отладки модуля WebAssembly.
- Из главы 13 вы узнаете, как писать интеграционные тесты для ваших модулей.

Каждая глава строится на том, что изложено в предыдущих главах, поэтому лучше читать их по порядку. Разработчикам нужно прочитать главы 1, 2 и 3 последовательно, чтобы понять, что такое WebAssembly, как он работает и как использовать набор инструментов Emscripten. Материалы приложения А также призваны помочь вам правильно установить эти средства, для того чтобы вы могли работать с примерами кода, представленными в этой книге. Первые две части книги освещают ключевые понятия. Оставшиеся части можно читать, исходя из ваших потребностей.

О КОДЕ

В этой книге содержится много примеров исходного кода, как в виде пронумерованных листингов, так и внутри обычного текста. Чтобы отличить его от нормального текста, код отформатирован с помощью `вот такого моноширинного шрифта`. Если код из предыдущего примера изменился, то отличия будут выделены **полужирным**.

В некоторых случаях код, демонстрируемый в книге, переформатирован с помощью разрывов строк и добавления отступов, чтобы использовать доступное место на странице. В редких случаях, где места все же не хватает, в коде будет стоять знак продолжения строки (➥).

Исходный код, представленный в этой книге, можно скачать с сайта издательства www.manning.com/books/webassembly-in-action.

ДРУГИЕ ОНЛАЙН-РЕСУРСЫ

Вам нужна дополнительная помощь?

- Много документации Emscripten доступно на сайте https://emscripten.org.
- Сообщество Emscripten очень активно, релизы выходят часто. Если ваша проблема связана с самим Emscripten, то можете посмотреть, не отправил ли кто-нибудь отчет об ошибке и не знает ли он, как решить эту проблему: https://github.com/emscripten-core/emscripten.
- Stack Overflow — тоже очень полезный сайт, на котором можно задать вопросы или помочь другим: https://stackoverflow.com/questions.

Об авторе

Жерар Галлан — сертифицированный специалист Microsoft и старший разработчик программного обеспечения в Dovico Software. Автор популярных блогов на Blogger.com и DZone.com.

Об иллюстрации на обложке

Обложку украшает рисунок *Fille Lipparotte* («Девушка из Липпаротт») из книги Жака Грассе де Сан-Савье (Jacques Grasset de Saint-Sauveur) *Costumes Civils Actuels de Tous les Peuples Connus* («Наряды из разных стран»), опубликованной во Франции в 1788 году. Широкое разнообразие коллекции нарядов Грассе де Сан-Савье напоминает нам о том, насколько 200 лет назад регионы мира были уникальны и индивидуальны. В те времена по одежде человека можно было легко определить, откуда он, чем занимается и каков его социальный статус.

Стиль изменился с тех пор, и столь богатое разнообразие различных регионов сгладилось. Зачастую непросто отличить даже жителей разных континентов, не говоря уже о городах, регионах и странах. Возможно, мы пожертвовали культурным многообразием в пользу большего разнообразия личной жизни — и определенно более разносторонней и динамичной жизни технологической.

В наше время, когда книги по информационным технологиям так мало отличаются друг от друга, издательство Manning отдает должное изобретательности и инициативности компьютерного бизнеса, создавая обложки книг, на которых отражено богатое разнообразие жизни регионов два века назад, и возвращая к жизни рисунки Жака Грассе де Сан-Савье.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I

Первые шаги

В этой части книги я познакомлю вас с WebAssembly и процессом создания модуля WebAssembly.

Из главы 1 вы узнаете, что такое WebAssembly, какие проблемы он решает, что обеспечивает его безопасность и какие языки программирования можно использовать для его создания.

В главе 2 я расскажу о внутренней структуре модуля WebAssembly, чтобы вы увидели, каково назначение каждого раздела.

Затем в главе 3 вы узнаете о различных параметрах вывода, доступных с помощью набора инструментальных средств Emscripten, и создадите первые модули WebAssembly. Кроме того, я расскажу об API WebAssembly на JavaScript.

1 Знакомство с WebAssembly

В этой главе

- ✓ Что такое WebAssembly.
- ✓ Проблемы, которые решает WebAssembly.
- ✓ Как работает WebAssembly.
- ✓ Что обеспечивает безопасность WebAssembly.
- ✓ Языки, с помощью которых можно создать модуль WebAssembly.

Во время веб-разработки для большинства разработчиков важнейшим фактором является производительность — насколько быстро загружается веб-страница и насколько быстро она откликается в целом. Многие исследования показали: если страница не загружается за три секунды, то 40 % посетителей уходят. Этот процент растет с каждой дополнительной секундой, уходящей на загрузку страницы.

Длительность загрузки веб-страницы не единственная проблема. В одной статье Google говорится следующее: если у страницы плохая производительность, то 79 % посетителей утверждают, что вряд ли будут приобретать что-либо на этом сайте повторно (Дэниел Ан и Пэт Минан, «Почему для маркетологов важна скорость загрузки мобильной страницы», июль 2016 г., http://mng.bz/MOlD).

Развитие веб-технологий породило тенденцию переносить все больше приложений в Интернет. Это поставило перед разработчиками новую сложную

задачу, поскольку браузеры поддерживают только один язык программирования: JavaScript.

С одной стороны, один язык программирования на все браузеры — это в чем-то хорошо: нужно написать код один раз, и вы уже знаете, что он запустится во всех браузерах. Вам в любом случае надо тестировать каждый браузер, который вы собираетесь поддерживать, поскольку их производители иногда реализуют одни и те же функции по-разному. Также иногда какой-то браузер не предоставляет новую функцию, которую уже имеют другие браузеры. Но в целом поддерживать один язык проще, чем четыре или пять. Недостаток того, что браузеры поддерживают только JavaScript, заключается в том, что приложения, которые мы хотим перенести в Интернет, написаны не на JavaScript — а, скорее всего, на таких языках, как C++.

JavaScript — прекрасный язык программирования, но теперь мы хотим, чтобы он выполнял не то, для чего был изначально разработан, — например, объемные вычисления для игр — и к тому же работал очень быстро.

1.1. ЧТО ТАКОЕ WEBASSEMBLY

Когда создатели браузеров начали искать способы улучшить производительность JavaScript, компания Mozilla (выпускающая браузер Firefox) определила подмножество JavaScript под названием asm.js.

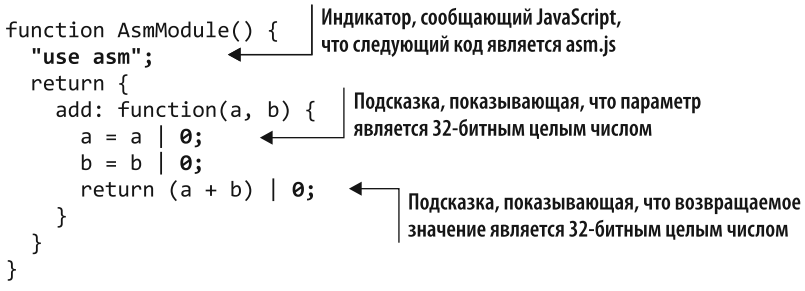
1.1.1. Asm.js, предшественник WebAssembly

Asm.js имел следующие преимущества.

- Asm.js не надо писать напрямую. Вместо этого вы пишете код на C или C++ и конвертируете его в JavaScript. Конвертация кода с одного языка на другой известна как *транспиляция (transpiling)*.
- Ускорение выполнения кода для сложных вычислений. Когда движок JavaScript видит особую строку под названием «инструкция `asm pragma`» (`"use asm";`), она действует как индикатор, сообщая браузеру, что он может использовать низкоуровневые системные операции вместо более сложных операций JavaScript.
- Ускорение выполнения кода с самого первого вызова. В asm.js включены подсказки при вводе кода, сообщающие JavaScript, какой тип данных будет содержать переменная. Например, `a | 0` будет использоваться как подсказка, что переменная `a` будет содержать 32-битное целое значение. Это работает, поскольку побитовая операция ИЛИ с нулем не меняет исходного значения, поэтому побочных эффектов нет.

Эти подсказки служат в качестве обещания (промиса, promise) для движка JavaScript, обозначая, что если код объявляет переменную как целое число, то она никогда не изменится, например, на строку. Следовательно, движку JavaScript не нужно отслеживать код, чтобы понять, какие где сейчас типы. Он может просто компилировать код в том виде, в каком он описан.

В следующем листинге показан пример кода asm.js:



Несмотря на все преимущества asm.js, у него есть и недостатки.

- Все эти подсказки могут значительно увеличить объем файлов.
- Файл asm.js является файлом на JavaScript, поэтому он все равно должен быть прочитан и разобран движком JavaScript. Это становится проблемой на таких устройствах, как телефоны, поскольку вся эта обработка замедляет время загрузки и тратит заряд батареи.
- Чтобы добавить новые возможности, создателям браузеров пришлось бы модифицировать сам язык JavaScript, что нежелательно.
- JavaScript является языком программирования и не создавался для того, чтобы быть целевым форматом для компиляции.

1.1.2. От asm.js к MVP

Подумав о том, как можно улучшить asm.js, создатели браузеров разработали минимальный прототип WebAssembly (minimum viable product, MVP), предназначенный для сохранения положительных аспектов asm.js и устранения его недостатков. В 2017 году все четыре основные компании, выпускающие браузеры (Google, Microsoft, Apple и Mozilla), добавили в свои браузеры поддержку MVP, также иногда называемого Wasm.

- WebAssembly — низкоуровневый язык наподобие ассемблера, который может запускаться на скорости, близкой к скорости нативного приложения, во всех современных «настольных» браузерах, а также и во многих мобильных браузерах.

- Файлы WebAssembly разработаны компактными, поэтому быстро передаются и загружаются. Вдобавок они разработаны так, чтобы их можно было быстро разобрать и инициализировать.
- WebAssembly разработан в качестве целевого формата для компиляции, чтобы код, написанный на таких языках, как C++, Rust и др., мог запускаться в веб-среде.

Разработчики серверной части приложений с помощью WebAssembly могут повысить повторное использование кода или перенести его в веб-среду, не переписывая. Кроме того, веб-разработчики извлекают выгоду из создания новых библиотек, улучшений существующих библиотек и возможности улучшить производительность сложных для вычисления фрагментов их собственного кода. Несмотря на то что основная площадка для использования WebAssembly — браузеры, он разработан переносимым, поэтому может применяться и вне браузера.

1.2. КАКИЕ ПРОБЛЕМЫ РЕШАЕТ WEBASSEMBLY

MVP WebAssembly направлен на решение следующих проблем asm.js.

1.2.1. Улучшение производительности

Одна из основных проблем, которые должен решить WebAssembly, — производительность: начиная от того, сколько времени требуется на загрузку вашего кода, заканчивая тем, насколько быстро он выполняется. Используя языки программирования, вместо того чтобы писать код на машинном языке, понятном компьютерному процессору (1 и 0, или нативный код), вы обычно пишете что-то близкое к человеческому языку. Проще работать с кодом, абстрагированным от конкретных деталей вашего компьютера, однако компьютерные процессоры его не понимают, поэтому, когда приходит время его запустить, нужно конвертировать написанное в машинный код.

JavaScript является так называемым *интерпретируемым языком программирования*, то есть он читает написанный вами код в процессе выполнения и переводит эти команды в машинный код по ходу работы. При использовании интерпретируемых языков не нужно компилировать код заранее, поэтому он начнет работать скорее. Но минус в том, что данный язык должен конвертировать команды в машинный код при каждом запуске кода. Если ваш код является циклом, например, то при каждом выполнении этого цикла нужно переводить каждую строку цикла. Поскольку на процесс интерпретации не всегда отводится большое количество времени, оптимизации тоже не всегда возможны.

Другие языки программирования, например C++, не являются интерпретируемыми. При использовании таких языков нужно конвертировать команды в машинный код заранее с помощью специальных программ под названием «компиляторы». В этом случае конвертация команд в машинный код занимает некоторое количество времени, прежде чем их можно будет запускать, но преимущество заключается в том, что отводится больше времени для выполнения оптимизации кода. После компиляции в машинный код его больше не надо будет компилировать.

Со временем JavaScript перестал быть просто языком-связкой, соединяющим компоненты между собой, с коротким жизненным циклом. Теперь его используют многие сайты для сложной обработки, часто с применением сотен и тысяч строк кода; и после того, как одностраничные приложения стали популярными, теперь этот код может иметь длительный жизненный цикл. Интернет прошел путь от сайтов, показывавших просто текст и пару картинок, до интерактивных веб-сайтов и даже тех, которые называются веб-приложениями, поскольку они похожи на «настольные» приложения, но запускаются в браузере.

Пока программисты продолжали проверять границы возможностей JavaScript, были обнаружены заметные проблемы с производительностью. Создатели браузеров решили попытаться найти золотую середину, которая позволила бы получить преимущества интерпретируемого языка, когда код начинает запускаться сразу после вызова, но при этом работает быстрее при выполнении. Чтобы ускорить код, создатели браузеров ввели понятие «JIT-компиляция» (just-in-time), когда движок JavaScript отслеживает код при его работе. Если фрагмент кода использовался достаточное количество раз, то движок попытается скомпилировать этот фрагмент в машинный код, чтобы в обход JavaScript-движка он использовал низкоуровневые системные вызовы, которые намного быстрее.

Движку JavaScript нужно выполнить код несколько раз, прежде чем он скомпилируется в машинный код, поскольку JavaScript является также динамическим языком программирования. В JavaScript переменная может содержать любой тип значения. Например, изначально она может содержать целое число, а затем за ней закрепят строку. Пока код не будет запущен несколько раз, браузер не знает, чего ожидать. Даже после компиляции за кодом по-прежнему надо следить, поскольку есть вероятность, что что-нибудь изменится и скомпилированный код для этого раздела нужно будет выкинуть и начать процесс заново.

1.2.2. Более короткое время запуска в сравнении с JavaScript

Как и asm.js, WebAssembly разработан не для того, чтобы его писали вручную, и не для того, чтобы его читал человек. Когда код скомпилирован в WebAssembly,

итоговый байт-код представлен не в текстовом, а в бинарном формате, что уменьшает размер файла и позволяет быстро передавать и загружать его.

Двоичный файл разработан таким образом, что валидацию модуля можно сделать за один проход. Вдобавок структура файла позволяет параллельно компилировать разные разделы этого файла.

С помощью применения JIT-компиляции создатели браузеров значительно улучшили производительность JavaScript. Но движок JavaScript может скомпилировать его в машинный код, только выполнив код несколько раз. С другой стороны, код WebAssembly статически типизирован, то есть типы значений, содержащихся в переменных, известны заранее. Поэтому код WebAssembly можно компилировать в машинный код с самого начала, не изучая его, — улучшения производительности заметны с первого запуска кода.

За время, прошедшее с первого релиза MVP, создатели браузеров нашли способы улучшить производительность WebAssembly. Одним из них было создание так называемой *потоковой компиляции* — это процесс компиляции кода WebAssembly в машинный код в процессе загрузки и получения файла браузером. Потоковая компиляция позволяет модулю WebAssembly инициализироваться, как только он загрузится, что значительно ускоряет время его запуска.

1.2.3. Возможность использовать в браузере другие языки, помимо JavaScript

До этого момента, чтобы использовать в Интернете язык, не являющийся JavaScript, код нужно было конвертировать в JavaScript, который не был разработан как целевой формат для компиляции. С другой стороны, WebAssembly был изначально разработан, чтобы быть целью компиляции, поэтому разработчики, желающие применять конкретный язык для веб-разработки, смогут это делать, не транспилируя свой код в JavaScript.

WebAssembly не привязан к языку JavaScript, поэтому его проще улучшать, не переживая о помехах работе JavaScript. Данная независимость должна привести к возможности быстрее улучшать WebAssembly. Основными языками WebAssembly MVP были выбраны C и C++, поскольку они могут компилироваться в WebAssembly, но с тех пор в язык Rust тоже была добавлена его поддержка, и несколько других языков начали экспериментировать с WebAssembly.

1.2.4. Возможность повторного использования кода

Возможность задействовать код, написанный не на JavaScript, и компилировать его в WebAssembly дает разработчикам больше простора при повторном

использовании кода. Теперь то, что раньше нужно было переписывать на JavaScript, можно использовать на обычном ПК или на сервере, или запускать в браузере.

1.3. КАК РАБОТАЕТ WEBASSEMBLY

Как показано на рис. 1.1, при использовании JavaScript код включен в сайт и переводится в процессе запуска. Поскольку переменные в JavaScript динамические, то по функции `add` на иллюстрации неочевидно, с каким типом переменных мы столкнулись. Переменные `a` и `b` могут быть целыми числами, числами с плавающей запятой, строками или даже комбинацией, в которой одна переменная может быть строкой, а другая, например, числом с плавающей запятой.

Единственный способ точно узнать эти типы — изучать код в процессе выполнения, как это делает движок JavaScript. Когда движок считает, что понял типы переменных, он может конвертировать данный фрагмент кода в машинный код.

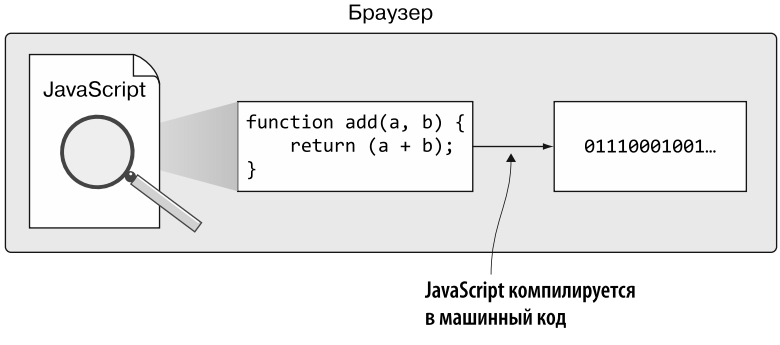


Рис. 1.1. JavaScript компилируется в машинный код в процессе выполнения

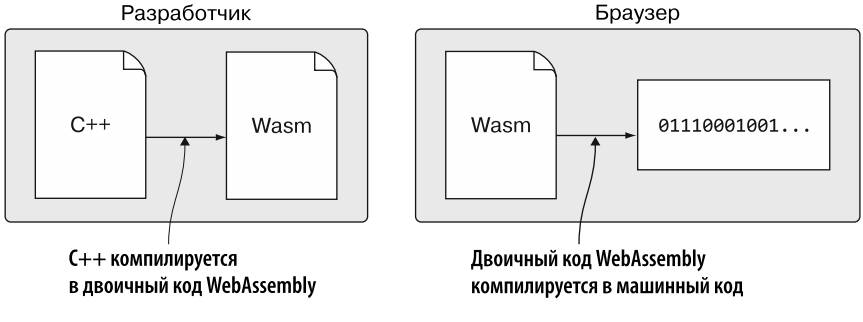


Рис. 1.2. C++ переводится в WebAssembly, а затем в машинный код в браузере

WebAssembly не интерпретируется, а вместо этого компилируется в двоичный формат WebAssembly разработчиком заранее, как показано на рис. 1.2. Поскольку все типы переменных известны заранее, то, когда браузер загружает файл WebAssembly, движку JavaScript не нужно изучать этот код. Он может просто скомпилировать двоичный формат в машинный код.

1.3.1. Обзор работы компиляторов

В подразделе 1.2.1 мы кратко поговорили о том, что разработчики пишут код на языке, близком к человеческому, а компьютерные процессоры понимают только машинный язык. В итоге написанный код нужно конвертировать в машинный, чтобы он выполнялся. Я не сказал о том, что у каждого типа компьютерного процессора свой тип машинного кода.

Было бы неэффективно компилировать каждый язык программирования напрямую в каждую версию машинного кода. Вместо этого обычно происходит то, что изображено на рис. 1.3: часть компилятора, называемая *фронтендом*, конвертирует написанный вами код в промежуточное представление (intermediate representation, IR). После того как код IR будет создан, *бэкенд*-часть компилятора берет его, оптимизирует и превращает в нужный машинный код.

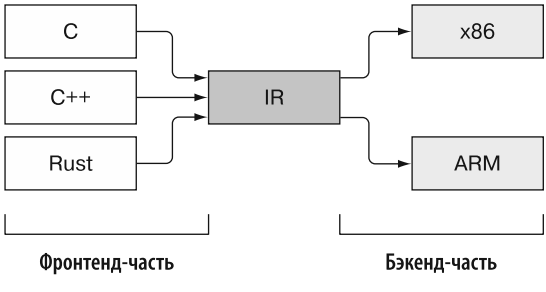


Рис. 1.3. Фронтенд- и бэкенд-части компилятора

Поскольку один и тот же браузер может работать на различных процессорах (от настольных компьютеров до смартфонов и планшетов), распространять скомпилированную версию кода WebAssembly для каждого потенциального процессора было бы слишком утомительно. На рис. 1.4 показано, что нужно сделать вместо этого: взять код IR и запустить его в специальном компиляторе, конвертирующем его в особый двоичный байт-код и сохраняющем в файл с расширением `.wasm`.

Байт-код в файле Wasm — еще не машинный код. Это просто набор виртуальных команд, понятный браузерам, поддерживающим WebAssembly. Как показано на рис. 1.5, когда файл загружен в браузер, поддерживающий WebAssembly, тот

проверяет, все ли с этим кодом в порядке, затем байт-код компилируется дальше в машинный код устройства, на котором запущен этот браузер.

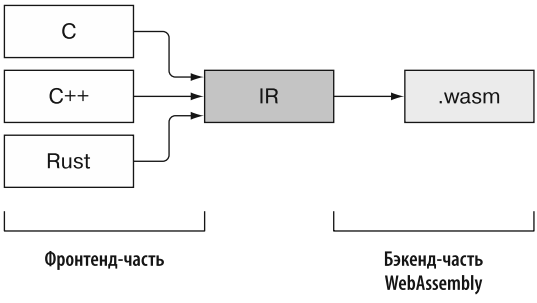


Рис. 1.4. Фронтенд-часть компилятора с бэкенд-частью для WebAssembly

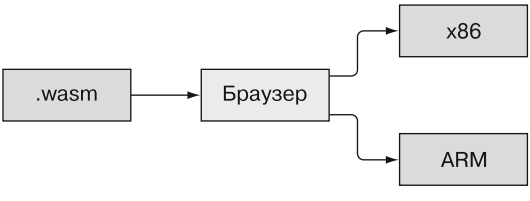


Рис. 1.5. Файл Wasm загружается в браузер и затем компилируется в машинный код

1.3.2. Загрузка, компиляция и создание модуля

В момент написания этой книги процесс загрузки файла Wasm в браузер и его компиляции производится с помощью вызовов функций JavaScript. В будущем разработчики хотят позволить модулям WebAssembly взаимодействовать с модулями ES6, благодаря чему станет возможно загружать модули WebAssembly через специальный HTML-тег (`<script type="module">`), но пока эта возможность недоступна. (ES — это сокращение от ECMAScript, а 6 — номер версии. ECMAScript — официальное название JavaScript.)

Перед тем как скомпилировать двоичный байт-код модуля WebAssembly, его нужно валидировать. Это позволит убедиться, что модуль правильно структурирован, код не выполняет ничего запрещенного и не имеет доступа к памяти, которая должна быть недоступна для модуля. Во время выполнения также делаются проверки на то, что данный код остается в пределах памяти, к которой у него есть доступ. Файл Wasm структурирован таким образом, чтобы валидация происходила за один проход, поскольку валидация, компиляция в машинный код и создание экземпляра модуля должны проходить как можно быстрее.

После того как браузер скомпилировал байт-код WebAssembly в машинный код, скомпилированный модуль можно передать в веб-воркер (мы изучим их более подробно в главе 9, но на данный момент нужно знать, что это способ создавать потоки в JavaScript) или в другое окно браузера. Скомпилированный модуль также может использоваться для создания дополнительных экземпляров этого модуля.

После компиляции файла Wasm нужно создать экземпляр модуля, прежде чем его использовать. *Создание экземпляра* — это просто процесс получения необходимых элементов импорта, инициации элементов модуля, вызова стартовой функции, если она была определена, и затем возвращения экземпляра модуля в среду выполнения.

WEBASSEMBLY ПРОТИВ JAVASCRIPT

До сих пор единственным языком, работавшим внутри виртуальной машины JavaScript (virtual machine, VM), был сам JavaScript. За прошедшие годы были испробованы многие технологии, например плагины, но для них требовалась собственная виртуальная машина-«песочница», что увеличивало и поверхность атаки, и использование ресурсов компьютера. Впервые VM JavaScript позволила коду WebAssembly работать на той же самой VM. У этого факта есть несколько преимуществ. Одно из важнейших из них заключается в том, что эта VM была тщательно протестирована и проверена на отсутствие уязвимых мест в системе безопасности на протяжении многих лет. Если бы нужно было создавать новую VM, то у нее, несомненно, были бы проблемы с безопасностью, которые пришлось бы решать.

WebAssembly разрабатывается как дополнение, а не замена JavaScript. Несмотря на возможность когда-нибудь увидеть разработчиков, которые попытаются создавать целые сайты, используя только WebAssembly, вряд ли это станет нормой. Будут моменты, когда JavaScript по-прежнему будет лучшим вариантом. Но будут и моменты, когда сайту понадобится WebAssembly, чтобы получить доступ к более быстрым вычислениям или поддержку на более низком уровне. Например, SIMD (single instruction, multiple data) — возможность обрабатывать много потоков данных с помощью одного потока команд — встраивалась в JavaScript в некоторых браузерах, но их создатели решили отменить ее реализацию на JavaScript и сделать ее доступной только через модули WebAssembly. В итоге, если вашему сайту нужна поддержка SIMD, то необходимо использовать модуль WebAssembly.

При программировании для браузера у вас обычно есть два основных компонента: VM JavaScript, в которой запущен модуль WebAssembly, и веб-API (например, DOM, WebGL, веб-воркеры и т. д.). Поскольку WebAssembly — это MVP, в нем недостает некоторых функций. Ваш модуль WebAssembly может связываться с JavaScript, но пока не может общаться напрямую ни с одним веб-API. Сейчас

идет работа над функциями пост-MVP, которая даст WebAssembly прямой доступ к веб-API. А пока модули могут взаимодействовать с веб-API в обход — вызывая JavaScript и заставляя его выполнять необходимое действие от имени модуля.

1.4. СТРУКТУРА МОДУЛЯ WEBASSEMBLY

Сейчас у WebAssembly есть только четыре доступных типа значений:

- 32-битные целые числа;
- 64-битные целые числа;
- 32-битные числа с плавающей запятой;
- 64-битные числа с плавающей запятой.

Булевы значения представлены с помощью 32-битного целого числа, где `0` — `false`, а значение `nonzero` — `true`. Все остальные типы значений, такие как строки, необходимо представлять в линейной памяти модуля.

Основная единица программы WebAssembly называется *модулем* — этот термин используется и для бинарной версии кода, и для скомпилированной версии в браузере. Вы не должны создавать модули WebAssembly вручную, но базовое понимание того, как структурирован модуль и как он работает на внутреннем уровне, может быть полезным, поскольку вы столкнетесь с определенными его аспектами при инициализации и во время жизненного цикла модуля.

На рис. 1.6 показано базовое представление структуры файла WebAssembly. Подробнее о структуре модуля вы узнаете в главе 2, а пока я предоставлю вам краткий обзор.

Файл Wasm начинается с раздела под названием *«преамбула»*.

1.4.1. Преамбула

Преамбула содержит «волшебное число» (`0x00 0x61 0x73 0x6D`, означающее `\0asm`), которое отличает модуль WebAssembly от модуля ES6. За этим волшебным числом следует номер версии (`0x01 0x00 0x00 0x00`, означающее `1`), указывающий, с помощью какой версии бинарного формата WebAssembly был создан данный файл.

На данный момент существует только одна версия бинарного формата. Одной из целей WebAssembly является сохранение обратной совместимости по мере добавления новых функций, а также стремление не допустить увеличения номера версии. Если когда-либо появится функция, которую нельзя будет применить, ничего не сломав, то номер версии будет увеличен.

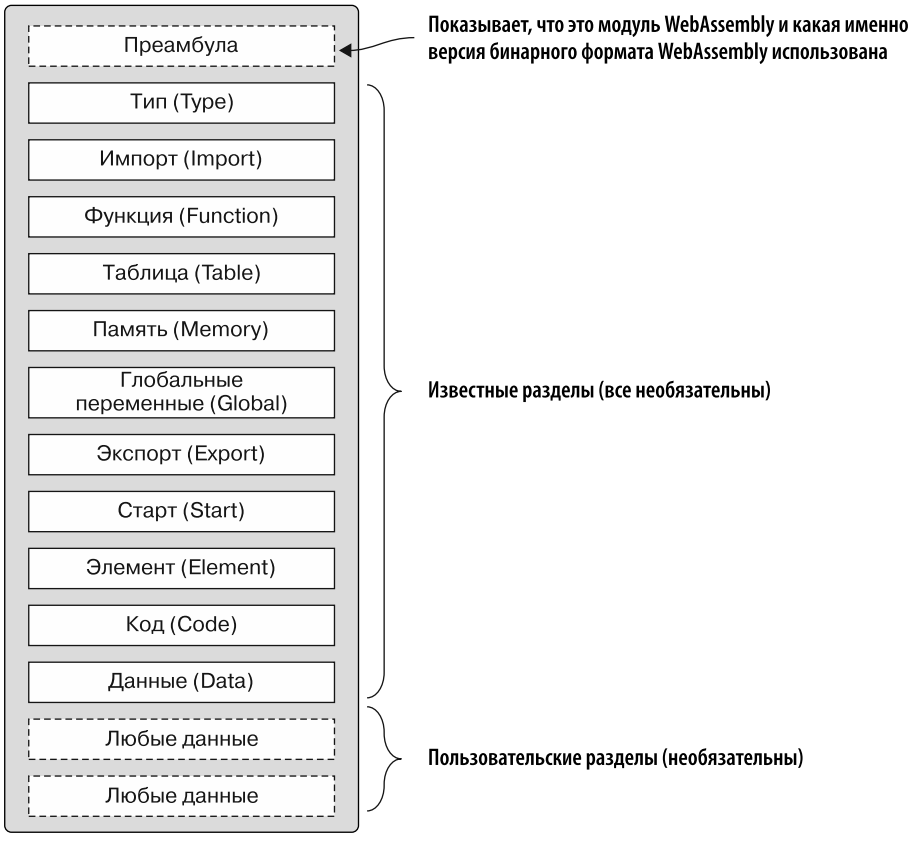


Рис. 1.6. Базовое представление структуры файла WebAssembly

За преамбулой идут несколько *разделов* модуля, но каждый из них необязателен, поэтому технически у вас может быть пустой модуль без разделов. Вы узнаете об одном варианте использования пустого модуля в главе 3, когда будете применять определение функций, чтобы проверить, поддерживается ли WebAssembly браузером.

Доступны два типа разделов: *известные* и *пользовательские*.

1.4.2. Известные разделы

Известные разделы могут входить в модуль всего один раз и должны появляться в определенном порядке. У каждого известного раздела есть конкретное назначение, он четко определен и валидируется в момент создания модуля. Эти разделы подробнее описываются в главе 2.

1.4.3. Пользовательские разделы

Пользовательские разделы предоставляют возможность внести в модуль данные для вариантов использования, неприменимых к известным разделам. Они могут появляться в любом месте модуля (до, между или после известных разделов) любое количество раз, и несколько пользовательских разделов могут даже задействовать одно и то же название.

В отличие от известных разделов, неправильно размещенный пользовательский раздел не вызовет ошибку подтверждения. Пользовательские разделы могут медленно загружаться фреймворком, поэтому содержащиеся в них данные могут быть недоступны до определенного момента после инициализации модуля.

В MVP WebAssembly был определен пользовательский раздел «Название» (Name). Идея этого раздела состоит в том, что у вас может быть отладочная версия вашего модуля и данный раздел может содержать названия функций и переменных в текстовой форме для использования при отладке. В отличие от других пользовательских разделов он должен появляться только один раз и лишь после раздела «Данные».

1.5. ТЕКСТОВЫЙ ФОРМАТ WEBASSEMBLY

WebAssembly разрабатывался с учетом открытости Интернета. Тот факт, что двоичный формат не предназначен для написания или чтения человеком, не означает, что модули WebAssembly являются способом скрыть код разработчиков. В действительности все наоборот. Для WebAssembly был определен текстовый формат, использующий *s-выражения* и соответствующий бинарному формату.

СПРАВКА

Символические выражения, или s-выражения, были изобретены для языка программирования Lisp. S-выражение может быть неделимой единицей или упорядоченной парой s-выражений, позволяющей вам формировать группу s-выражений. Неделимая единица — это символ, не являющийся списком: например, foo или 23. Список представлен скобками и может быть пустым или содержать неделимые единицы или даже другие списки. Каждая единица разделена пробелами: например, (), или (foo), или (foo (bar 132)).

Этот текстовый формат предполагается использовать, например, для функции просмотра исходного кода в браузере или для отладки. Вы даже можете писать s-выражения от руки и с помощью специального компилятора переводить код в двоичный формат WebAssembly.

Текстовый формат WebAssembly будет использоваться браузерами, когда вы решите просмотреть исходник, и в целях отладки, поэтому вам пригодится базовое понимание данного формата. Например, поскольку все разделы модуля необязательны, то вы можете определить пустой модуль с помощью следующего s-выражения:

```
(module)
```

Если вы хотите скомпилировать это s-выражение в двоичный формат WebAssembly и посмотреть на полученные бинарные значения, то файл будет содержать только байты из начальной части: `0061 736d` (волшебное число) и `0100 0000` (номер версии).

ЗАБЕГАЯ ВПЕРЕД

В главе 11 вы создадите модуль WebAssembly, используя только текстовый формат, чтобы лучше понимать увиденное, если вам, например, понадобится отладить модуль в браузере.

1.6. КАК ОБЕСПЕЧИВАЕТСЯ БЕЗОПАСНОСТЬ WEBASSEMBLY

Безопасность WebAssembly обеспечивается за счет того, что это первый язык, работающий в VM JavaScript, изолированной от среды выполнения. Виртуальную машину укрепляли и тестировали годами, чтобы обеспечить ее безопасность. У модулей WebAssembly нет доступа к тому, что недоступно для JavaScript, и они следуют той же политике безопасности, включающей в себя в том числе политику единого источника.

В отличие от приложения для рабочего стола у модуля WebAssembly нет прямого доступа к памяти устройства. Вместо этого среда выполнения передает модулю объект `ArrayBuffer` при инициализации. Модуль использует этот `ArrayBuffer` в качестве линейной памяти, а фреймворк WebAssembly проверяет, точно ли код работает в границах массива.

У модуля WebAssembly нет прямого доступа к объектам, хранящимся в разделе «Таблица», например к указателям на функции. Код просит у фреймворка WebAssembly доступ к объекту, основанный на его индексе. Затем фреймворк обращается к памяти и выполняет объект от имени этого кода.

В C++ стек выполнения находится в памяти вместе с линейной памятью, и хотя код на C++ не должен модифицировать данный стек, это возможно при использовании указателей. Стек выполнения WebAssembly тоже отделен от линейной памяти и недоступен для кода.

ИНФОБОКС

Если вас интересует дополнительная информация о модели безопасности WebAssembly, то можете посетить следующий сайт: https://webassembly.org/docs/security.

1.7. КАКИЕ ЯЗЫКИ МОЖНО ИСПОЛЬЗОВАТЬ ДЛЯ СОЗДАНИЯ МОДУЛЯ WEBASSEMBLY

При создании MVP WebAssembly был изначально сосредоточен на языках C и C++, но с тех пор его поддержка появилась и в Rust, и в AssemblyScript. Кроме того, написать код можно, используя текстовый формат WebAssembly с s-выражениями и скомпилировав это в WebAssembly специальным компилятором.

Сейчас у MVP WebAssembly нет сборки мусора (garbage collection, GC), что ограничивает возможности некоторых языков. GC разрабатывается в качестве функции пост-MVP, но до ее появления некоторые языки экспериментируют с WebAssembly, либо компилируя свои VM в WebAssembly, либо, в некоторых случаях, включая собственную сборку мусора.

Следующие языки экспериментируют с WebAssembly или поддерживают его:

- C и C++;
- Rust нацелен на то, чтобы быть языком программирования по выбору для WebAssembly;
- AssemblyScript — новый компилятор, который конвертирует TypeScript в WebAssembly. Это имеет смысл, учитывая, что TypeScript типизирован и уже транспилируется в JavaScript;
- TeaVM — инструмент, транспилирующий Java в JavaScript, теперь он может также генерировать WebAssembly;
- Go 1.11 добавил экспериментальный порт в WebAssembly, включающий в себя сборку мусора в качестве части скомпилированного модуля;
- Pyodide — это порт Python, включающий в себя основные пакеты научного стека Python: Numpy, Pandas и matplotlib;
- Blazor — экспериментальная попытка Microsoft перенести C# в WebAssembly.

ИНФОБОКС

Хранилище GitHub, размещенное на https://github.com/appcypher/awesome-wasm-langs, поддерживает тщательно отобранный список языков, которые компилируются в WebAssembly или имеют в нем виртуальные машины. Вдобавок в данном списке указано, на каком этапе поддержки WebAssembly находится тот или иной язык.

Для обучения WebAssembly в этой книге мы используем C и C++.

1.8. ГДЕ МОЖНО ИСПОЛЬЗОВАТЬ МОДУЛЬ

В 2017 году все современные создатели браузеров выпустили версии, поддерживающие MVP WebAssembly. В этот список входят Chrome, Edge, Firefox, Opera и Safari. Некоторые мобильные браузеры также поддерживают WebAssembly, в том числе Chrome, Firefox для Android, а также Safari.

Как уже упоминалось в начале главы, WebAssembly был разработан с учетом переносимости, чтобы его можно было использовать в разных локациях, а не только в браузере. Сейчас разрабатывается новый стандарт WASI (WebAssembly Standard Interface, Стандартный интерфейс WebAssembly), который обеспечит устойчивую работу модулей WebAssembly во всех поддерживаемых системах. Статья Лин Кларк «Стандартизация WASI: системный интерфейс для запуска WebAssembly за пределами браузера» (27 марта 2019 года) содержит хороший обзор WASI и доступна на http://mng.bz/gVJ8.

ИНФОБОКС

Если вам хочется подробнее узнать о WASI, обратитесь в хранилище GitHub по адресу https://github.com/wasmerio/awesome-wasi — там есть тщательно отобранный список связанных с ним ссылок и статей.

Одной из небраузерных локаций, поддерживающих модули WebAssembly, является Node.js, начиная с версии 8. Node.js — это среда выполнения JavaScript, собранная с помощью движка JavaScript Chrome V8, позволяющего использовать код на JavaScript в серверной части. Многие разработчики видят в WebAssembly возможность применять в браузере знакомый им код вместо JavaScript. Аналогично и Node.js позволяет разработчикам, предпочитающим JavaScript, использовать его в серверной части. Чтобы продемонстрировать применение WebAssembly вне браузера, в главе 10 мы покажем вам, как работать с модулем WebAssembly в Node.js.

WebAssembly не замена JavaScript, а, скорее, его дополнение. Иногда лучше выбрать модуль WebAssembly, но временами лучше задействовать JavaScript. Работа на одной виртуальной машине с JavaScript позволяет обеим технологиям пользоваться преимуществами друг друга.

WebAssembly предоставит разработчикам, профессионально пишущим не на JavaScript, возможность сделать свой код доступным в Сети. Он также позволит веб-разработчикам, которые могут не уметь писать код на таких языках, как C и C++, получить доступ к более новым и быстрым библиотекам и потенциально к тем, чьи функции недоступны в текущих библиотеках для JavaScript. В некоторых случаях модули WebAssembly могут использоваться библиотеками, чтобы ускорить выполнение определенных аспектов последних, — за исключением более быстрого кода, библиотека будет работать так же, как и всегда.

Самое потрясающее в WebAssembly — то, что он уже доступен во всех основных «настольных» браузерах, в нескольких основных мобильных браузерах и даже вне браузера в Node.js.

РЕЗЮМЕ

Как вы увидели в этой главе, WebAssembly значительно улучшает производительность, а также выбор языка и повторное использование кода. Ниже изложены некоторые ключевые моменты, улучшенные с помощью WebAssembly.

- Передача и загрузка ускоряются благодаря уменьшению размеров файлов в связи с использованием бинарного кода.
- Благодаря структуре файлов Wasm их можно быстро разбирать и валидировать. Вдобавок она позволяет параллельно компилировать различные части файла.
- При помощи потоковой компиляции модули WebAssembly можно компилировать в процессе загрузки, поэтому они готовы к созданию экземпляра модуля в момент завершения загрузки, что значительно сокращает время подготовки модуля.
- Код выполняется быстрее в таких случаях, как вычисления, в связи с использованием вызовов на уровне устройства вместо более медленных вызовов движка JavaScript.
- Не нужно изучать код перед компиляцией, чтобы определить, как он себя поведет. В результате код работает с одной и той же скоростью при каждом запуске.
- Поскольку WebAssembly отделен от JavaScript, то его можно быстрее улучшать, так как это не повлияет на язык JavaScript.
- Можно использовать в браузере код, написанный не на JavaScript.
- Возможность повторного использования кода увеличивается благодаря структурированию фреймворка WebAssembly таким образом, что он может применяться и в браузере, и вне его.

2 Модули WebAssembly изнутри

В этой главе

- ✓ Описание известных и пользовательских разделов модуля WebAssembly.

В этой главе вы узнаете о разных разделах модуля WebAssembly и их назначении. По мере чтения книги я буду открывать вам больше подробностей, но сейчас нужно получить базовое понимание того, как структурированы модули и как разные разделы работают вместе.

Ниже представлены некоторые преимущества наличия разных разделов в модуле и способов их разработки.

- *Эффективность* — двоичный байт-код можно разобрать, валидировать и скомпилировать за один проход.
- *Потоковость* — парсинг, валидация и компиляция могут начаться до окончания загрузки всех данных.
- *Параллелизация* — парсинг, валидация и компиляция могут производиться параллельно.
- *Безопасность* — у модуля нет прямого доступа к памяти устройства, и такие объекты, как указатели на функции, вызываются от имени вашего кода.

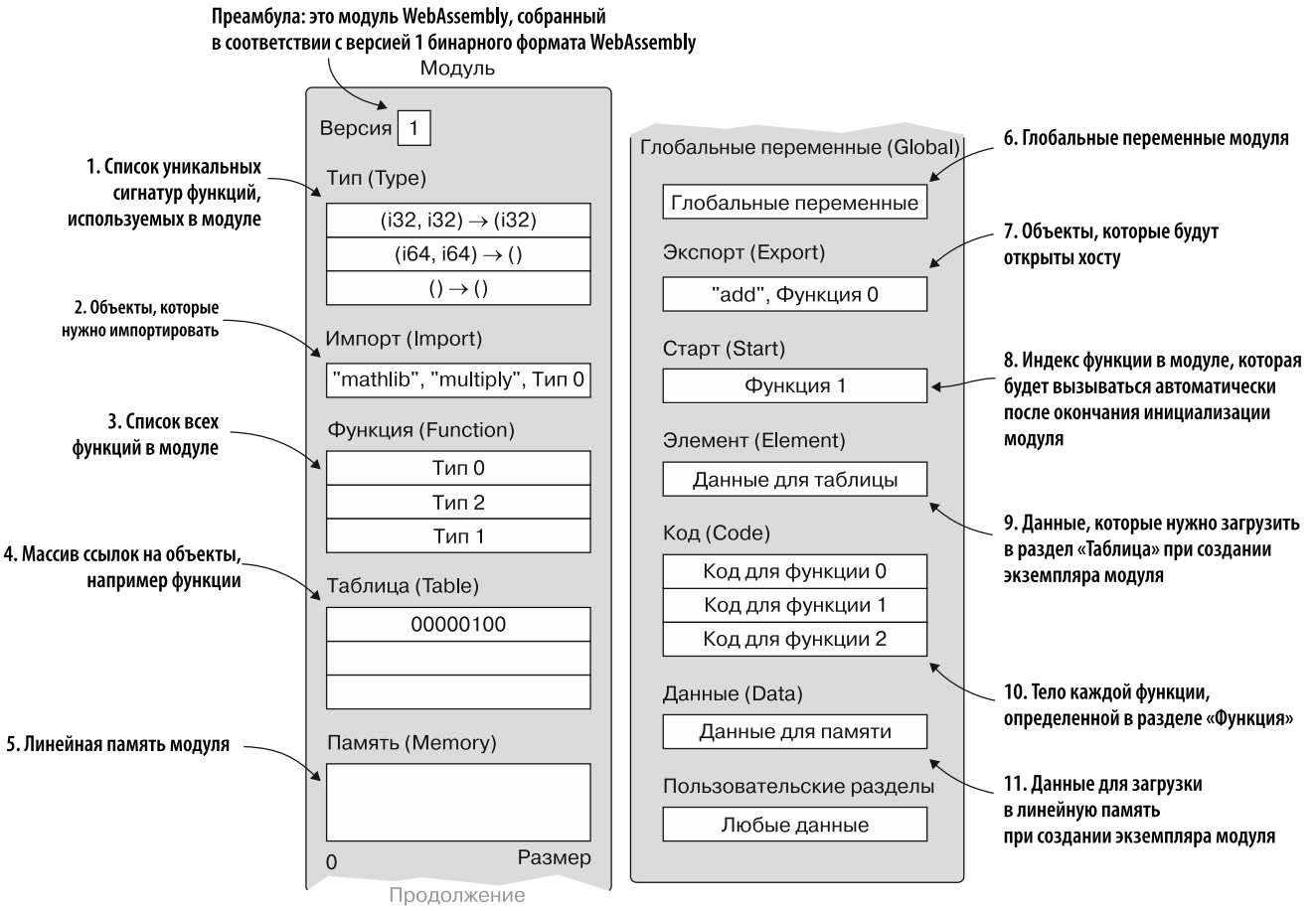


Рис. 2.1. Базовая структура бинарного байт-кода WebAssembly с выделением известных и пользовательских разделов

На рис. 2.1 представлена базовая структура бинарного байт-кода WebAssembly. Хотя вы будете взаимодействовать с разными разделами при работе с модулями, компилятор отвечает за создание необходимых разделов и размещение их в правильном порядке, основываясь на вашем коде.

Модули WebAssembly могут содержать несколько разделов, но каждый из них необязателен. Технически у вас может быть пустой модуль без разделов. Как уже было сказано в главе 1, существует два типа доступных разделов:

- известные разделы;
- пользовательские разделы.

Известные разделы имеют конкретную цель, точно определены и валидируются в момент создания экземпляра модуля WebAssembly. Пользовательские разделы используются для данных, которые не относятся к известным разделам и не вызывают ошибку валидации, если они неправильно размещены.

Байт-код WebAssembly начинается с преамбулы, указывающей на то, что это модуль WebAssembly и версия 1 бинарного формата WebAssembly. После преамбулы идут известные разделы, все из которых являются необязательными. На рис. 2.1 пользовательские разделы показаны в конце модуля, но в реальности их можно размещать перед известными разделами, между ними или после них. Как и известные, пользовательские разделы тоже необязательны.

Теперь, когда вы имеете общее представление о базовой структуре модуля WebAssembly, рассмотрим каждый из известных разделов поближе.

2.1. ИЗВЕСТНЫЕ РАЗДЕЛЫ

Если известный раздел включается в модуль, то это можно сделать максимум один раз. Эти разделы должны появляться в представленном здесь порядке.

Тип

Раздел *«Тип» (Type)* объявляет список всех уникальных сигнатур функций, которые будут использованы в модуле, включая те, что будут импортированы. Несколько функций могут иметь одну и ту же сигнатуру.

На рис 2.2 представлен пример раздела «Тип», содержащего три сигнатуры функций:

- в первой два 32-битных целочисленных (i32) параметра и 32-битное целочисленное (i32) возвращаемое значение;

- во второй два 64-битных целочисленных (i64) параметра, но нет возвращаемого значения;
- третья не принимает никаких параметров и не возвращает значение.

Импорт

Раздел *«Импорт» (Import)* заявляет обо всех случаях импортирования, которые будут использоваться в модуле и могут применяться в разделах «Функция», «Таблица», «Память» и «Глобальные переменные».

Импортирование разработано таким образом, чтобы модули могли иметь общий код и данные, но тем не менее могли компилироваться и кэшироваться по отдельности. Импортирование предоставляется средой хоста после того, как будет создан экземпляр модуля.

Функция

Раздел *«Функция» (Function)* — это список всех функций в модуле. Положение объявления функции в данном списке показывает индекс тела функции в разделе «Код». Значение, указанное в разделе «Функция», указывает на индекс сигнатуры функции в разделе «Тип».

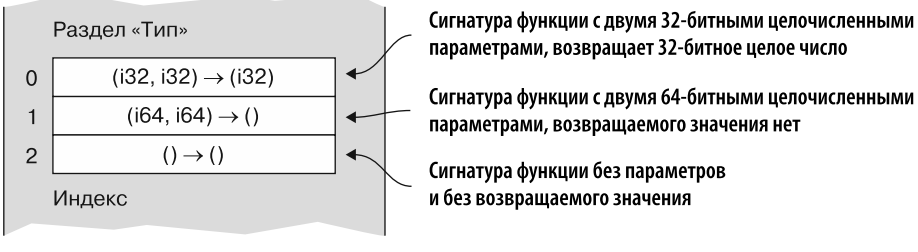


Рис. 2.2. Раздел «Тип», содержащий три сигнатуры функций. Сигнатура с индексом 0 получает два 32-битных целочисленных параметра и возвращает 32-битное целочисленное значение. Сигнатура с индексом 1 получает два 64-битных целочисленных параметра, но не имеет возвращаемого значения. Сигнатура с индексом 2 не получает никаких значений параметра и не имеет возвращаемого значения

На рис. 2.3 показан пример связи между разделами «Тип», «Функция» и «Код». Если взглянуть на раздел «Функция», то значение второй функции является индексом сигнатуры функции, у которой нет параметров и возвращаемого значения. Индекс второй функции указывает на совпадающий индекс в разделе «Код».

Объявление функции отделено от тела функции, чтобы можно было осуществлять параллельную и потоковую компиляцию каждой функции в модуле.

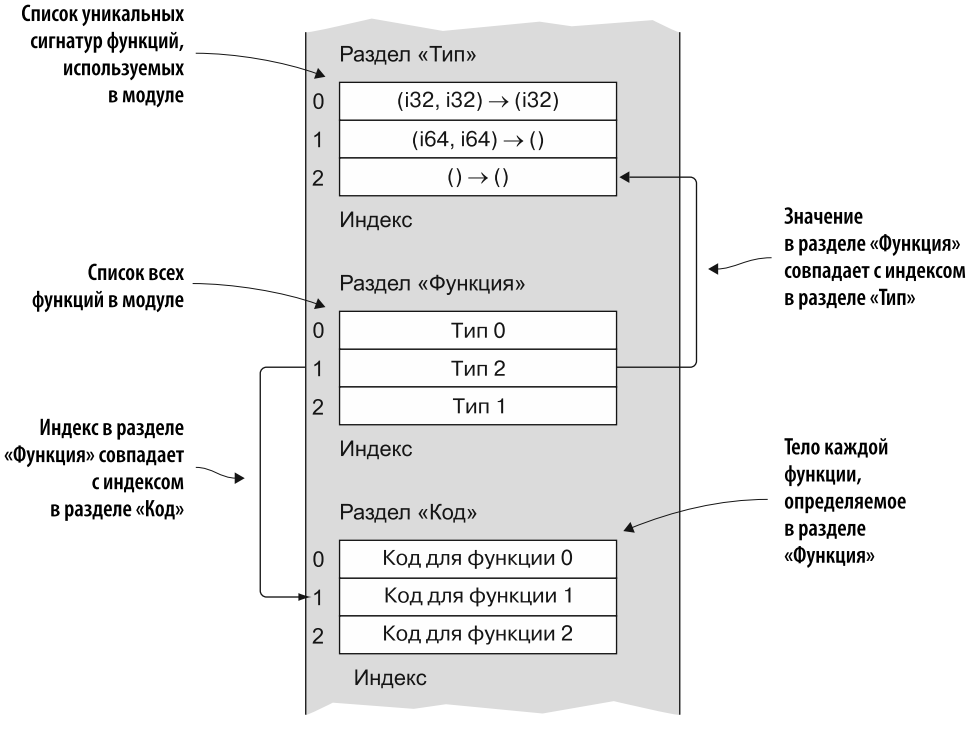


Рис. 2.3. Пример того, как связаны разделы «Тип», «Функция» и «Код»

Таблица

Раздел *«Таблица» (Table)* содержит типизированный массив ссылок, например на функции, которые не могут храниться в линейной памяти модуля в виде последовательности байтов. Этот раздел обеспечивает один из ключевых аспектов безопасности WebAssembly, предоставляя фреймворку WebAssembly способ безопасно отображать объекты.

У вашего кода нет прямого доступа к ссылкам, хранящимся в таблице. Вместо этого, когда ему нужен доступ к данным, указанным в этом разделе, он просит фреймворк произвести операцию с объектом с конкретным индексом в таблице. Фреймворк WebAssembly читает адрес, сохраненный в этом индексе, и производит действие. В частности, при работе с функциями это позволяет оперировать «указателями на функции», задавая при этом индекс в таблице.

На рис. 2.4 показан код WebAssembly, который просит вызвать объект с индексом 0 в разделе «Таблица». Фреймворк WebAssembly читает адрес ячейки памяти по этому индексу и затем выполняет код, размещенный с указанного места в памяти.

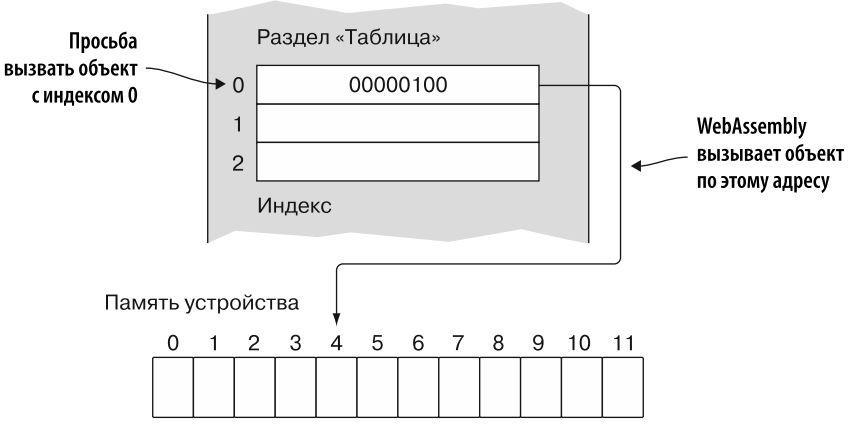


Рис. 2.4. Пример вызова объекта из раздела «Таблица»

Таблица определяется с начальным размером, и дополнительно может быть определен максимальный размер. Размер таблицы равен количеству элементов. Можно увеличить таблицу на заданное количество элементов. Если определено максимальное количество элементов, то система не даст таблице увеличиться дальше данной точки. Но если максимум не указан, то таблица может расти без ограничений.

Память

Раздел *«Память»* (*Memory*) хранит линейную память, используемую данным экземпляром модуля.

Раздел «Память» также является ключевым аспектом безопасности WebAssembly, поскольку у модулей WebAssembly нет прямого доступа к памяти устройства. Вместо этого, как показано на рис. 2.5, среда выполнения, создающая экземпляр модуля, передает `ArrayBuffer`, который используется в качестве линейной памяти данным экземпляром. Пока это касается кода, данная линейная память работает как область динамической памяти в C++, но каждый раз, когда код пытается получить доступ к этой памяти, фреймворк проверяет, находится ли данный запрос в границах массива.

Память модуля определяется в страницах WebAssembly, каждая из которых имеет размер 64 Кбайт (1 Кбайт равен 1024 байтам, поэтому в одной странице 65 536 байт). Когда среда выполнения указывает, сколько памяти может быть у модуля, она указывает начальное количество страниц и дополнительно максимальное количество страниц. Если модулю нужно больше памяти, то можно запросить увеличение памяти на указанное количество страниц. Если указано максимальное количество страниц, то фреймворк не даст памяти увеличиться

дальше этой точки. Если же оно не указано, то память может расти без ограничений.

Рис. 2.5. ArrayBuffer используется модулями WebAssembly как линейная память

Несколько экземпляров модулей WebAssembly могут разделять одну общую линейную память (`ArrayBuffer`), что полезно, если модули динамически связаны.

В C++ стек выполнения находится в памяти вместе с линейной памятью. Хотя код на C++ не должен модифицировать стек выполнения, это можно сделать с помощью указателей. Помимо того что у кода нет доступа к памяти устройства, WebAssembly добавил еще один уровень безопасности и отделил стек выполнения от линейной памяти.

Глобальные переменные

В разделе *«Глобальные переменные» (Global)* находятся определения глобальных переменных модуля.

Экспорт

В разделе *«Экспорт» (Export)* содержится список всех объектов, которые будут предоставлены хосту после того, как будет создан экземпляр модуля (фрагменты модуля, к которым среда хоста имеет доступ). Это может включать экспортирование разделов «Функция», «Таблица», «Память» и «Глобальные переменные».

Старт

Раздел *«Старт» (Start)* задает индекс функции, которая будет вызвана после инициализации модуля, но перед тем, как могут быть вызваны экспортированные функции. Стартовая функция может использоваться как способ инициализации глобальных переменных или памяти. Функция не может быть импортирована, если это указано. Она должна существовать внутри модуля.

Элемент

Раздел *«Элемент» (Element)* объявляет данные, которые загружаются в раздел «Таблица» при создании экземпляра модуля.

Код

Раздел *«Код» (Code)* содержит тело каждой функции, заявленной в разделе «Функция». Все тела функций должны появляться в том же порядке, в каком они были объявлены. (См. описание связи между разделами «Тип», «Функция» и «Код» на рис. 2.3.)

Данные

Раздел *«Данные» (Data)* содержит данные, которые будут загружены в линейную память модуля при создании экземпляра.

В главе 11 вы узнаете о текстовом формате WebAssembly, который является текстовым эквивалентом бинарного формата модуля. Он используется браузерами для отладки модуля, если карты кода недоступны. Текстовый формат также может пригодиться, если нужно проинспектировать сгенерированные вами модули, чтобы посмотреть, как их создал компилятор, и найти причины неполадок. В текстовом формате используются те же названия разделов, которые вы узнали в этой главе, но иногда они отображаются в сокращенном виде (например, `func` вместо «Функция»).

Модуль также может включать пользовательские разделы, позволяющие добавить в него данные, не относящиеся к известным разделам, описанным в этой главе.

2.2. ПОЛЬЗОВАТЕЛЬСКИЕ РАЗДЕЛЫ

Пользовательские разделы могут появляться в любом месте модуля (перед известными разделами, между ними или после них) любое количество раз. Несколько пользовательских разделов даже могут иметь одно и то же название.

В отличие от известных разделов неправильное размещение пользовательского раздела не вызовет ошибку валидации. Пользовательские разделы могут медленно загружаться фреймворком, поэтому содержащиеся в них данные могут быть недоступны до какого-то момента после инициализации модуля.

Один из вариантов применения пользовательских разделов — раздел «Название» (Name), определенный для MVP WebAssembly. В нем можно размещать названия функций и переменных в текстовой форме, чтобы облегчить отладку. Но в отличие от нормальных пользовательских разделов он может появляться только один раз, если его решили включить, и лишь после известного раздела «Данные».

РЕЗЮМЕ

В данной главе вы узнали об известных и пользовательских разделах модуля WebAssembly и получили представление о том, за что они отвечают и как связаны. Это поможет вам при взаимодействии с модулями WebAssembly и работе с текстовым форматом WebAssembly. В частности, вы узнали следующее.

- Разделы модуля WebAssembly и то, как они спроектированы, связаны со многими возможностями и преимуществами WebAssembly.
- Компилятор отвечает за генерацию разделов модуля и размещение их в правильном порядке.
- Все разделы необязательны, поэтому возможно существование пустого модуля.
- Если известные разделы определены, то они могут появляться только один раз и в конкретном порядке.
- Пользовательские разделы могут размещаться до известных разделов, между ними и после них и служат для размещения данных, не относящихся к известным разделам.

3 Создание вашего первого модуля WebAssembly

В этой главе

- ✓ Обзор набора инструментальных средств Emscripten.
- ✓ Создание модуля с помощью Emscripten и HTML-шаблона Emscripten.
- ✓ Создание модуля с использованием связующего кода Emscripten JavaScript и загрузка модуля с помощью этого кода.
- ✓ Создание модуля без использования связующего кода Emscripten JavaScript и загрузка модуля самостоятельно.
- ✓ Определение возможностей с целью проверить доступность WebAssembly.

В текущей главе вы напишете код на C и затем примените набор инструментальных средств Emscripten, чтобы скомпилировать его в модуль WebAssembly. Это позволит нам рассмотреть три подхода к данному набору инструментов, с помощью которых мы можем создавать модули WebAssembly. Чтобы дать вам представление о том, что позволяет сделать этот набор, в WebAssembly с помощью Emscripten были перенесены некоторые объекты, в том числе Unreal Engine 3, SQLite и AutoCAD.

3.1. НАБОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ EMSCRIPTEN

Набор инструментальных средств Emscripten на данный момент самый развитый из доступных инструментариев для компиляции кода на C или C++ в байт-код WebAssembly. Изначально он был создан для транспиляции такого кода в asm.js. Когда началась работа над MVP WebAssembly, Emscripten был выбран, поскольку он использует компилятор LLVM, а рабочая группа WebAssembly уже имела опыт работы с LLVM при разработке Google Native Client (PNaCl). Emscripten по-прежнему подходит для транспиляции кода на C и C++ в asm.js, но вы будете применять его для компиляции написанного вами кода в модули WebAssembly.

Как описано в главе 1, у компиляторов обычно есть фронтенд-часть, которая конвертирует исходный код в промежуточное представление (IR), и бэкенд-часть, предназначенная для конвертации IR в нужный машинный код, как показано на рис. 3.1.

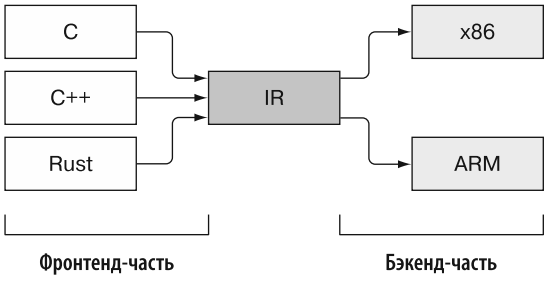


Рис. 3.1. Фронтенд-часть и бэкенд-часть компилятора

Я уже упоминал, что Emscripten задействует компилятор LLVM — инструментальные средства этого компилятора на данный момент лучше всех поддерживают WebAssembly, к тому же он удобен тем, что к нему можно подключить несколько фронтенд- и бэкенд-частей. Компилятор Emscripten использует Clang, похожий на GCC в C++, в качестве компилятора фронтенд-части для конвертации кода на C или C++ в IR LLVM, как показано на рис. 3.2. Затем Emscripten конвертирует IR LLVM в двоичный байт-код, представляющий собой виртуальный набор инструкций, понятных браузерам, поддерживающим WebAssembly. Поначалу это может звучать немного пугающе, но, как вы увидите в данной главе, процесс компиляции кода на C или C++ в модуль WebAssembly — это простая команда в окне консоли.

Прежде чем продолжить, пожалуйста, ознакомьтесь с приложением A, чтобы установить Emscripten, и убедитесь в наличии у вас всех инструментов, которые понадобятся в этой книге. Установив их, можете переходить к следующему разделу.

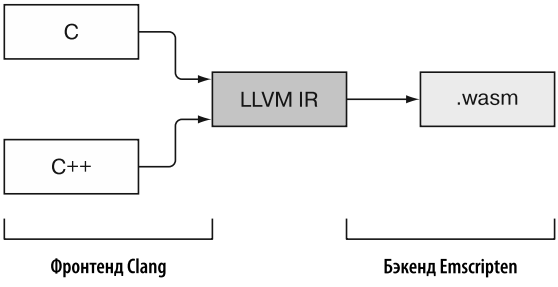


Рис. 3.2. Фронтенд компилятора C IR LLVM

3.2. МОДУЛИ WEBASSEMBLY

Когда файл WebAssembly загружается браузером, поддерживающим WebAssembly, браузер проводит проверку с целью убедиться, что все в правильном порядке. Если файл прошел проверку, то браузер скомпилирует байт-код в машинный код устройства, как показано на рис. 3.3.

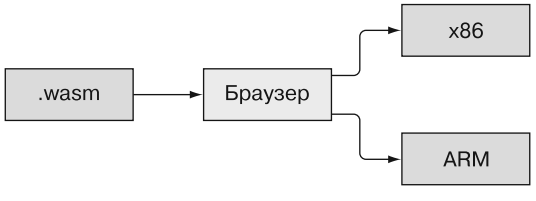


Рис. 3.3. Файл WebAssembly загружается в браузер и компилируется в машинный код

И двоичный файл WebAssembly, и скомпилированный объект в браузере называются *модулями*. Хотя вы можете создать и пустой модуль, он не будет особенно полезен, поэтому у большинства модулей есть хотя бы одна функция, производящая некую обработку. Функции модулей могут быть встроенными, импортированными из экспорта других модулей или даже импортированными из JavaScript.

В модулях WebAssembly есть несколько разделов, которые Emscripten наполнит, основываясь на вашем коде на C или C++. С точки зрения внутреннего устройства, разделы начинаются с ID раздела, за которым идет его размер и затем само содержимое. Подробную информацию о разделах модуля можно найти в главе 2. Все разделы необязательны, поэтому модуль и может оказаться пустым.

Раздел «Старт» указывает на индекс функции, являющейся частью модуля (неимпортированной). Указанная функция будет автоматически вызвана прежде, чем любой экспортированный объект сможет быть вызванным JavaScript. Если включить в код на C или C++ функцию `main`, то Emscripten установит ее в качестве стартовой функции модуля.

Модуль WebAssembly получает используемую память от хоста в форме `ArrayBuffer`. В том, что касается модуля, буфер ведет себя как область динамической памяти в C или C++, но каждый раз, когда модуль взаимодействует с памятью, фреймворк WebAssembly проверяет, находится ли запрос в границах массива.

Модули WebAssembly поддерживают только четыре типа данных:

- 32-битные целые числа;
- 64-битные целые числа;
- 32-битные числа с плавающей запятой;
- 64-битные числа с плавающей запятой.

Булевы значения представлены с помощью 32-битного целого числа, где `0` — `false`, а значение `nonzero` — `true`. Все остальные типы значений, определенные средой хоста, такие как строки, необходимо представлять в линейной памяти модуля.

Файлы WebAssembly содержат двоичный байт-код, предназначенный не для чтения человеком, а для того, чтобы быть максимально эффективным и быстро загружаться, компилироваться и создавать экземпляр модуля. В то же время модули WebAssembly не предназначены быть «черными ящиками», в которых разработчики могут скрыть свой код. WebAssembly был разработан с учетом открытости Интернета, поэтому его двоичный формат имеет эквивалентное представление в текстовом формате. Последний можно увидеть, если зайти в инструменты разработчика в браузере.

Модули WebAssembly имеют несколько преимуществ.

- Они созданы для того, чтобы быть целевым форматом при компиляции, в отличие от JavaScript. Это позволит в будущем улучшать WebAssembly, не затрагивая JavaScript.
- Они разработаны переносимыми, то есть их можно использовать не только в браузерах. Сейчас модули WebAssembly можно применять и в Node.js.
- Файлы WebAssembly используют двоичный формат, поэтому максимально компактны и их можно быстро передавать и загружать.
- Файлы структурированы таким образом, чтобы валидация происходила за один проход, что уменьшает время загрузки.
- С помощью последних функций API WebAssembly на JavaScript файл можно скомпилировать в машинный код в процессе загрузки, и он будет готов к использованию сразу после ее завершения.
- Из-за динамической природы JavaScript код нужно выполнить несколько раз, прежде чем скомпилировать его в машинный код. А байт-код WebAssembly

компилируется в машинный код сразу же. В итоге первый вызов функции имеет такую же скорость, как и, например, десятый.

- Благодаря предварительной компиляции компилятор может оптимизировать код еще до того, как он появится в браузере.
- Код WebAssembly запускается почти так же быстро, как нативный код. Поскольку WebAssembly проверяет, правильно ли работает код, наблюдается небольшое снижение производительности в сравнении с работой чисто нативного кода.

3.2.1. Когда не стоит использовать модуль WebAssembly

Несмотря на то что WebAssembly имеет много преимуществ, он не всегда является нужным вариантом. В некоторых обстоятельствах лучше выбрать JavaScript:

- если код простой, то дополнительная работа по установке инструментальных средств компилятора и написанию на другом языке будет просто пустой тратой сил и времени;
- хотя сейчас над этим вопросом идет работа и все может измениться, но в данный момент у модулей WebAssembly нет прямого доступа к DOM или любым веб-API.

ОПРЕДЕЛЕНИЕ

DOM, или объектная модель документа, — это интерфейс, который отображает различные аспекты веб-страницы, предоставляя коду на JavaScript возможность взаимодействовать со страницей.

3.3. ПАРАМЕТРЫ ВЫВОДА EMSCRIPTEN

Модули WebAssembly можно создавать разными способами в зависимости от ваших целей. Можно дать Emscripten команду сгенерировать файл модуля WebAssembly и, в зависимости от параметров, указанных в командной строке, Emscripten может также включить *связующий файл JavaScript* и файл HTML.

ОПРЕДЕЛЕНИЕ

Связующий файл JavaScript — это файл JavaScript, генерируемый Emscripten. Его содержимое может варьироваться в зависимости от аргументов командной строки. Данный файл содержит код, который будет автоматически загружать файл WebAssembly и обеспечивать его компиляцию и создание экземпляра модуля в браузере. JavaScript также содержит множество вспомогательных функций, чтобы хосту было проще взаимодействовать с модулем и наоборот.

Создать модуль с помощью Emscripten позволяют следующие три подхода.

- *Дать Emscripten команду сгенерировать модуль WebAssembly, связующий файл JavaScript и файл шаблона HTML.*

  Emscripten, создающий HTML-файл, — это нетипично для производственной системы, но это полезно, если вы изучаете WebAssembly и хотите сосредоточиться на компиляции C или C++, прежде чем погрузиться в детали того, что требуется для загрузки и создания экземпляра модуля. Вдобавок данный метод полезен, если вы хотите поэкспериментировать с фрагментами кода, чтобы отладить что-то или создать прототип. С помощью этого подхода вы можете просто написать код на C или C++, скомпилировать его и затем открыть сгенерированный HTML-файл в браузере, чтобы увидеть результаты.

- *Дать Emscripten команду сгенерировать модуль WebAssembly и связующий файл JavaScript.*

  В производственной системе обычно используется этот подход, поскольку можно добавить сгенерированный файл JavaScript к новой или уже существующей HTML-странице, просто включив в нее ссылку на данный файл. Этот файл JavaScript автоматически загрузится и создаст экземпляр модуля при загрузке HTML-страницы. У него также есть несколько вспомогательных функций, упрощающих взаимодействие между модулем и вашим JavaScript.

  И этот подход, и подход с шаблоном HTML будут включать в себя объекты любой стандартной библиотеки C, если ваш код будет их использовать. Если ваш код не задействует функцию стандартной библиотеки C, но вам нужно включить ее в модуль, то можете использовать параметры, чтобы Emscripten включил нужные вам функции.

- *Дать Emscripten команду сгенерировать только модуль WebAssembly.*

  Этот подход нужен для динамического связывания двух или более модулей в среде выполнения, но подойдет и для создания минималистического модуля без поддержки стандартных библиотек C и связующего файла JavaScript.

ОПРЕДЕЛЕНИЕ

Более подробно я изложу данную информацию в главах 7 и 8, но сейчас вам нужно знать, что динамическое связывание модулей WebAssembly — это процесс соединения двух или более модулей в среде выполнения, где нераспознанные символы в одном модуле (например, функция) разрешаются в символы, существующие в другом.

Если вашему коду нужно передавать между модулем и JavaScript что-то кроме целых чисел и чисел с плавающей запятой, то ему понадобится управление

памятью. Если у вас нет эквивалента из стандартной библиотеки для функций `malloc` и `free`, то я не рекомендую этот подход для данного сценария. Линейная память модуля в действительности представляет собой буфер на массиве, переданный модулю при создании экземпляра, поэтому проблемы с памятью не повлияют на браузер или ОС, но могут привести к ошибкам, которые будет сложно отследить.

Кроме случаев динамического связывания, этот подход полезен для изучения того, как вручную загрузить, скомпилировать и создать экземпляр модуля с помощью API WebAssembly на JavaScript, — повторить то, что может сделать за вас связующий код Emscripten. Знание того, что делают функции API WebAssembly на JavaScript, упростит понимание ряда примеров, которые можно найти в Сети.

Поскольку Emscripten не единственный доступный компилятор, который может создавать модули WebAssembly (например, такой компилятор есть и у Rust), то в будущем вы можете решить использовать сторонний модуль, не имеющий кода для загрузки. В какой-то момент вам может потребоваться вручную загрузить и создать экземпляр модуля.

3.4. КОМПИЛЯЦИЯ C ИЛИ C++ С ПОМОЩЬЮ EMSCRIPTEN И ШАБЛОНА HTML

Допустим, вас попросили написать код, который будет определять, какие простые числа существуют в определенном диапазоне чисел. Вы могли написать его на JavaScript, но прочитали, что одна из главных областей, где WebAssembly прекрасно проявляет себя, — это вычисления, и потому решили использовать его для данного проекта.

Вам нужно будет интегрировать проект в существующий сайт, но сначала вы хотите создать модуль WebAssembly, чтобы убедиться, что все работает должным образом. Вы напишете код на C, а затем компилируете его в модуль WebAssembly с помощью Emscripten. Очень удобно, что, как показано на рис. 3.4, Emscripten может генерировать JavaScript, необходимый для загрузки и компиляции модуля WebAssembly, а также создавать HTML-файлы из шаблонов.

Первое, что вам нужно сделать, — создать папку, в которой будут храниться ваши файлы: WebAssembly\Chapter 3\3.4 html_template\.

ПРИМЕЧАНИЕ

В этой книге используются обозначения Windows для разделителей файлов. Пользователям *nix нужно заменить символы \ на /.

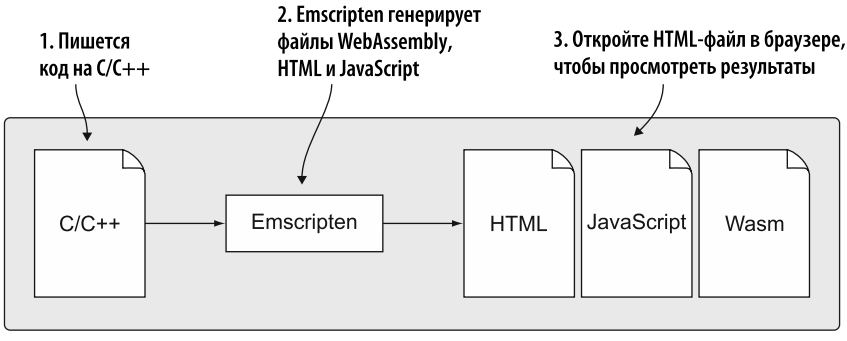


Рис. 3.4. Emscripten генерирует файлы WebAssembly, JavaScript и HTML

Как показано на рис. 3.5, первый шаг процесса — создание кода на C или C++. Создайте файл `calculate_primes.c` и откройте его. Первое, что вам нужно будет сделать, — включить заголовочный файл для стандартной библиотеки C, стандартной библиотеки ввода-вывода C и библиотеки Emscripten:

```
#include <stdlib.h>
#include <stdio.h>
#include <emscripten.h>
```

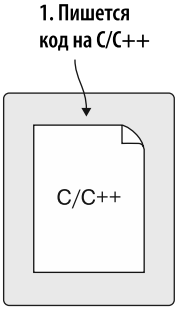
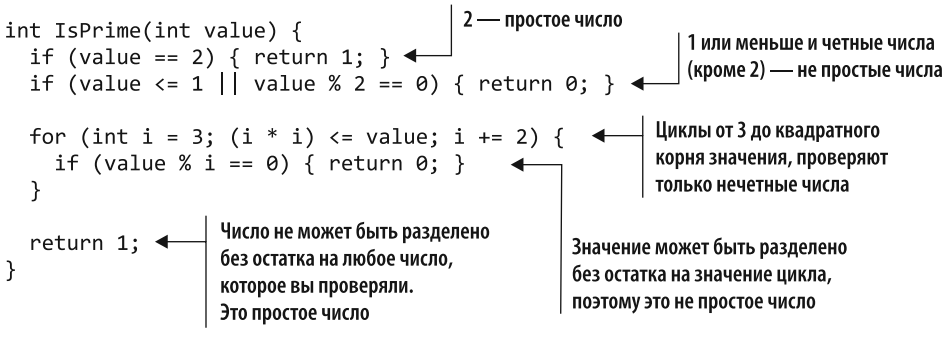


Рис. 3.5. Шаг 1 — создание кода на C или C++

Следующий шаг — написать вспомогательную функцию `IsPrime`, которая будет принимать в качестве параметра целочисленное значение, которое вы будете проверять, чтобы понять, простое ли это число. Если да, то функция будет возвращать `1`. Если нет, то будет возвращать `0` (ноль).

Простое число — это любое число, которое может делиться без остатка только на 1 и на само себя. За исключением 2, четные числа никогда не являются простыми, поэтому функция может их пропускать. Кроме того, поскольку проверка любого числа выше, чем квадратный корень из него, будет лишней, ваш

код может пропускать и эти числа, что сделает код чуть более эффективным. Основываясь на этом, вы можете создать в файле `calculate_primes.c` следующую функцию:



Теперь, когда у вас есть функция, которая может определить, является ли значение простым числом, нужно написать код, чтобы пройти в цикле по диапазону чисел, вызвать функцию `IsPrime` и вывести значение, если это простое число. Код для этих действий не требует никакого взаимодействия с JavaScript, поэтому включите его в функцию `main`. Видя функцию `main` в вашем коде на C или C++, Emscripten укажет ее в качестве стартовой функции модуля. Когда модуль загрузится и скомпилируется, фреймворк WebAssembly автоматически вызовет стартовую функцию.

Вы используете функцию `printf` в вашей функции `main`, чтобы передать строки в код Emscripten на JavaScript. Затем данный код отобразит полученные строки в текстовом поле на веб-странице и в окне консоли инструментов разработчика в браузере. В главе 4 вы напишете код, в котором модуль будет взаимодействовать с кодом на JavaScript, что позволит вам лучше разобраться в том, как устроено это взаимодействие.

После функции `IsPrime` можно написать код, показанный в листинге 3.1, чтобы пройти в цикле от 3 до 100 000 и найти среди этих чисел простые.

На рис. 3.6 показан следующий шаг этого процесса, когда вы командуете компилятору Emscripten конвертировать ваш код на C в модуль WebAssembly. В данном случае вам также нужно, чтобы Emscripten создал связующий файл на JavaScript и файл шаблона HTML.

Для компиляции кода на C в модуль WebAssembly нужно использовать окно консоли, чтобы запустить команду `emcc`, которая является компилятором Emscripten. Вместо того чтобы указывать путь к файлам, которые вы хотите скомпилировать, проще перейти в папку WebAssembly\Chapter 3\3.4 html_template\. Откройте окно консоли и перейдите туда.

Листинг 3.1. Функция main в файле calculate_primes.c

```
...

int main() {
  int start = 3;
  int end = 100000;

  printf("Prime numbers between %d and %d:\n", start, end);

  for (int i = start; i <= end; i += 2) {
    if (IsPrime(i)) {
      printf("%d ", i);
    }
  }
  printf("\n");

  return 0;
}
```

Начинается с нечетного числа, чтобы сделать эффективнее следующий цикл

Сообщает коду на JavaScript, какой диапазон

Проходит в цикле по диапазону чисел, проверяя только нечетные

Если текущее значение является простым числом, то сообщает его коду на JavaScript

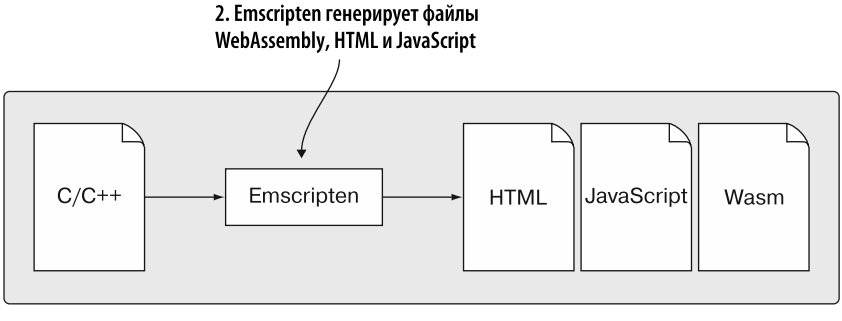


Рис. 3.6. Emscripten дана команда скомпилировать код на C в файл WebAssembly, сгенерировать связующий файл JavaScript и HTML-файл

Команда `emcc` принимает некоторое количество параметров и флагов. Хотя их порядок не имеет значения, входные файлы следует указывать в начале. В этом случае нужно поместить `calculate_primes.c` после `emcc`.

По умолчанию, если вы не указываете имя выходного файла, Emscripten не будет генерировать HTML-файл, а вместо этого сгенерирует файл WebAssembly `a.out.wasm` и файл JavaScript `a.out.js`. Чтобы указать выходной файл, нужно использовать параметр `-o` (дефис и строчная «о»), а затем желаемое название файла. Чтобы Emscripten создал шаблон HTML, нужно указать название файла с расширением `.html`.

Запустите следующую команду, чтобы сгенерировать модуль WebAssembly, который связывает файл JavaScript и шаблон HTML. Обратите внимание: это

может занять пару минут, если вы впервые запускаете компилятор Emscripten, поскольку он также будет создавать общие ресурсы, которые сможет использовать повторно. Они будут кэшированы, поэтому последующие компиляции будут проходить гораздо быстрее:

```
emcc calculate_primes.c -o html_template.html
```

ИНФОБОКС

Существует несколько флагов оптимизации, доступных на сайте Emscripten по адресу https://emscripten.org/docs/optimizing/Optimizing-Code.html. Emscripten рекомендует начинать без оптимизаций при первом портировании кода. Если не указывать флаг оптимизации, то в командной строке по умолчанию проставляется -O0 (заглавная «O» и ноль). Нужно отладить и исправить все проблемы, которые могут существовать в вашем коде, прежде чем начать включать оптимизации. Затем, в зависимости от ваших потребностей, можно будет применять флаги оптимизации от -O0 до -O1, -O2, -Os, -Oz и -O3.

Если вы посмотрите в папку, в которую поместили файл `calculate_primes.c`, то теперь должны увидеть еще три файла, выделенные на рис. 3.7.

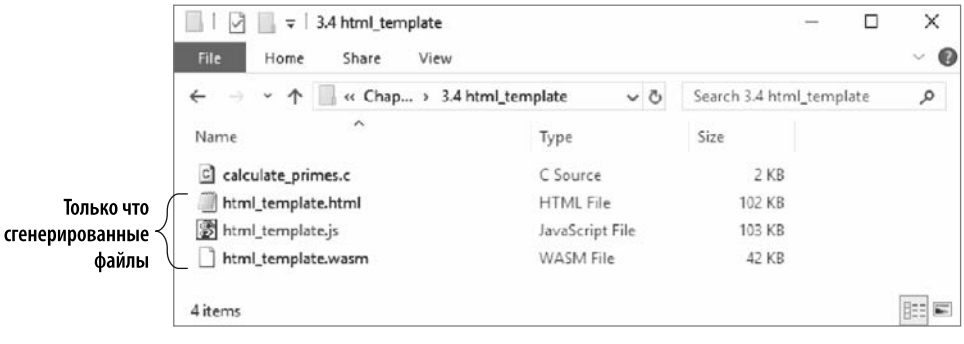


Рис. 3.7. Только что сгенерированные файлы HTML, JavaScript и WebAssembly

Файл `html_template.wasm` — это ваш модуль WebAssembly. Файл `html_template.js` — это сгенерированный файл JavaScript, а `html_template.html` — это ваш HTML-файл.

Как показано на рис. 3.8, финальный шаг этого процесса — просмотр веб-страницы для подтверждения того, что модуль WebAssembly ведет себя ожидаемым образом.

Если вы используете Python для локального веб-сервера, перейдите в папку WebAssembly\Chapter3 \3.4 html_template\ и запустите веб-сервер. Откройте браузер и введите в адресную строку следующий адрес: `http://localhost:8080/`

`html_template.html` (в зависимости от вашего сервера вам может не понадобиться фрагмент `:8080`).

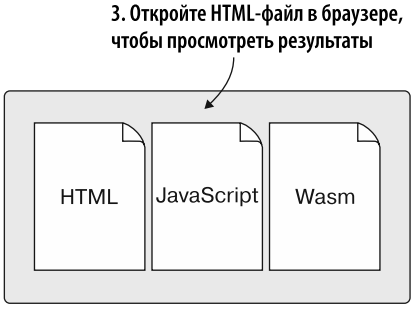


Рис. 3.8. Теперь можно открыть HTML-файл в браузере, чтобы просмотреть результаты

Вы увидите сгенерированную HTML-страницу, как показано на рис. 3.9.

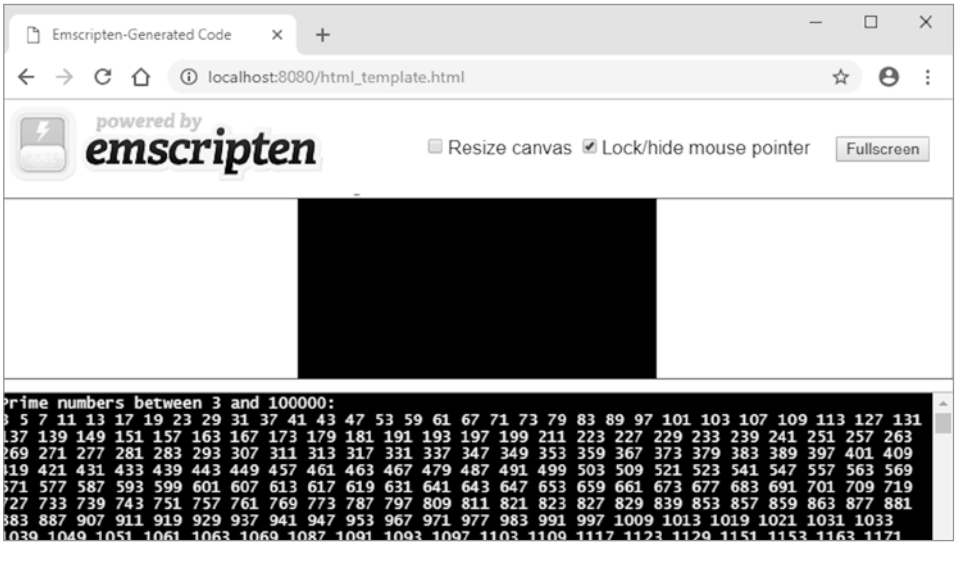


Рис. 3.9. HTML-страница, запущенная в Google Chrome

СОВЕТ

Чтобы установить набор средств Emscripten, необходимо установить еще и Python. Это удобно, поскольку Python также может запускать локальный веб-сервер. Если вы хотите использовать другой сервер для примеров из данной книги, то можете это сделать, но убедитесь, что на нем присутствует тип данных WebAssembly. Инструкции по запуску локального веб-сервера с помощью Python можно найти в приложении A. Тип данных, ожидаемый браузерами при загрузке модуля WebAssembly, также упоминается в этом приложении.

HTML-файл, созданный Emscripten, направляет любой вывод `printf` из модуля в текстовое поле, чтобы вы могли видеть его на странице, не открывая инструменты разработчика в браузере. Вдобавок в этот файл входит элемент «холст» (canvas) над текстовым полем для вывода WebGL. WebGL — это API, основанный на OpenGL ES 2.0, позволяющий веб-контенту отрисовывать 2D- и 3D-графику на холст.

В последующих главах вы узнаете, как Emscripten направляет вывод из вызова `prinf` в консоль отладчика в браузере или в текстовое поле.

3.5. EMSCRIPTEN ГЕНЕРИРУЕТ СВЯЗУЮЩИЙ КОД НА JAVASCRIPT

Возможность отдать Emscripten команду включить файл шаблона HTML может быть полезна, если вы хотите быстро проверить код или подтвердить, что с логикой модуля все в порядке, прежде чем продолжить. Но при использовании кода для реальных задач файл шаблона обычно не применяется. Вместо этого нужно дать Emscripten команду скомпилировать ваш код на C или C++ в модуль WebAssembly и сгенерировать связывающий файл JavaScript. Затем вы либо создаете новую веб-страницу, либо редактируете существующую и включаете в нее ссылку на файл JavaScript. Когда ссылка на файл становится частью веб-страницы, при загрузке страницы файл автоматически начинает загружать и создавать экземпляр модуля WebAssembly.

3.5.1. Компиляция кода на C или C++ вместе с JavaScript, созданного Emscripten

Вы проверили свой код для простых чисел, создав модуль WebAssembly с шаблоном HTML с помощью Emscripten. Теперь, когда код модуля готов и работает ожидаемым образом, вам нужно дать Emscripten команду сгенерировать только модуль WebAssembly и связующий файл JavaScript. Как показано на рис. 3.10, вы создадите ваш собственный файл HTML и затем добавите ссылку на сгенерированный файл JavaScript. Первое, что вам нужно сделать, — создать папку, в которой будут храниться файлы для этого подраздела: WebAssembly\Chapter 3\3.5 js_plumbing\.

Как показано на рис. 3.11, первый шаг — это создание кода на C или C++. Листинг 3.2 отображает содержимое файла `calculate_primes.c`, который вы создали для использования с шаблоном HTML. Скопируйте данный файл в вашу папку 3.5 js_plumbing.

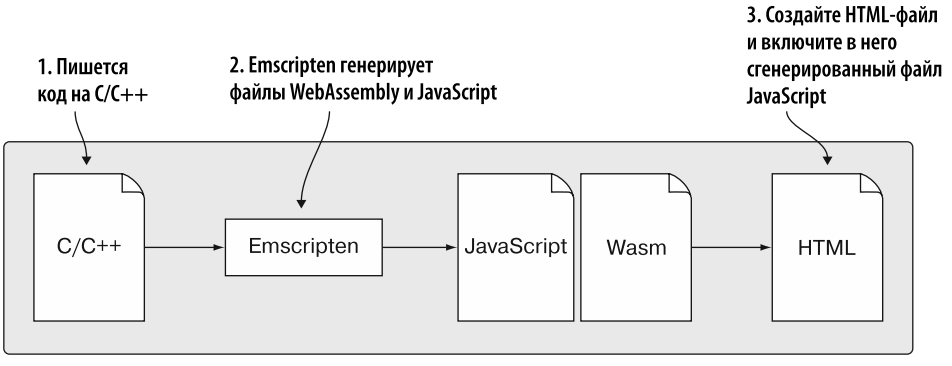


Рис. 3.10. Даете команду Emscripten сгенерировать файл WebAssembly и связующий файл JavaScript. Затем создаете HTML-файл и включаете в него ссылку на сгенерированный файл JavaScript

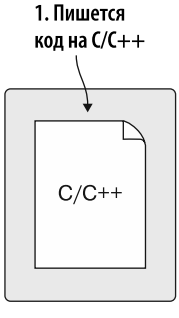


Рис. 3.11. Шаг 1 — создание кода на C или C++

Листинг 3.2. Код в файле calculate_primes.c

```
#include <stdlib.h>
#include <stdio.h>
#include <emscripten.h>

int IsPrime(int value) {
  if (value == 2) { return 1; }
  if (value <= 1 || value % 2 == 0) { return 0; }

  for (int i = 3; (i * i) <= value; i += 2) {
    if (value % i == 0) { return 0; }
  }

  return 1;
}

int main() {
  int start = 3;
  int end = 100000;
```

```
  printf("Prime numbers between %d and %d:\n", start, end);

  for (int i = start; i <= end; i += 2) {
    if (IsPrime(i)) {
      printf("%d ", i);
    }
  }
  printf("\n");

  return 0;
}
```

Теперь у вас есть новый файл на C. Следующий шаг показан на рис. 3.12 — дать компилятору Emscripten команду конвертировать ваш код на C в модуль WebAssembly. Вам также нужно, чтобы Emscripten включил связующий файл JavaScript, но не файл шаблона HTML.

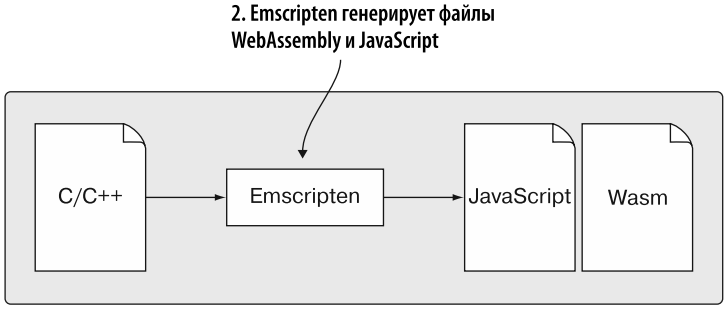


Рис. 3.12. Командуем Emscripten скомпилировать код на C в файл WebAssembly и сгенерировать связующий файл JavaScript

Чтобы скомпилировать ваш код на C в модуль WebAssembly, нужно открыть окно консоли и перейти в папку WebAssembly\Chapter 3\3.5 js_plumbing\. Здесь используется команда, похожая на ту, которую вы применили при включении шаблона HTML. В этом случае вы хотите сгенерировать только файлы WebAssembly и JavaScript. Вам не требуется HTML-файл, поэтому следует изменить название файла вывода, чтобы он имел расширение `.js` вместо `.html`. Запустите следующую команду, чтобы Emscripten создал модуль WebAssembly и файл JavaScript:

```
emcc calculate_primes.c -o js_plumbing.js
```

Если вы посмотрите в папку, в которую скопировали файл `calculate_primes.c`, то теперь должны увидеть два новых файла, выделенных на рис. 3.13.

Теперь у вас есть модуль WebAssembly и сгенерированный файл JavaScript. На рис. 3.14 показан следующий этап — создание HTML-файла и включение

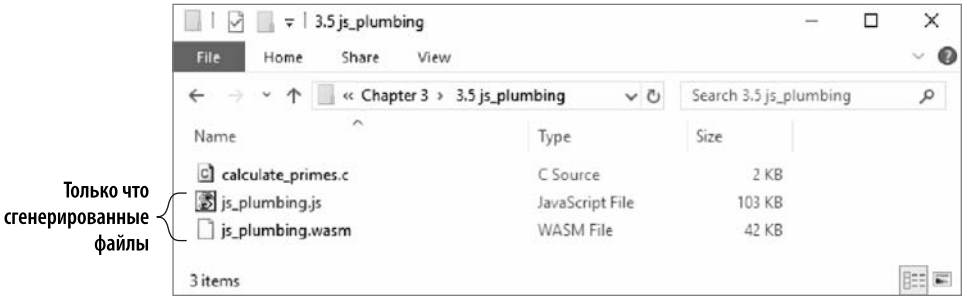


Рис. 3.13. Только что сгенерированные файлы JavaScript и WebAssembly

в него сгенерированного файла JavaScript. Данный файл управляет загрузкой и созданием экземпляра модуля WebAssembly, поэтому для получения доступа к функциям модуля нужно просто включить в HTML-страницу ссылку на файл.

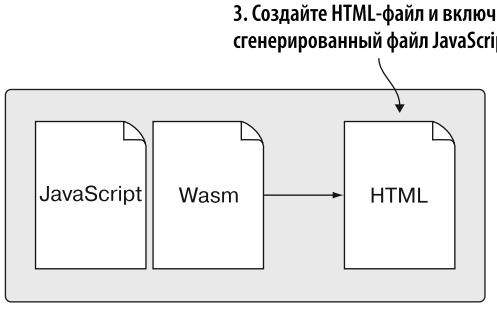


Рис. 3.14. HTML-файл изменен или создан новый, чтобы добавить в него ссылку на сгенерированный файл JavaScript

3.5.2. Создание базовой HTML-страницы для использования в браузерах

Для разработчиков, хорошо владеющих такими языками, как C или C++, но никогда по-настоящему не работавших с HTML-страницами, я кратко расскажу об элементах страниц, которые вы сможете быстро создать, чтобы использовать для примеров из этой главы. Если вы уже разбираетесь в основах создания HTML-страниц, то можете перейти к следующему пункту, «Создание HTML-страницы».

Основы HTML

Первое, что нужно для каждой HTML-страницы, — это объявление `DocType`, сообщающее браузеру, какая версия HTML используется. Последней версией является HTML 5, советую использовать ее. `DocType` для HTML 5 пишется таким образом: `<!DOCTYPE html>`.

По большей части HTML — это серия тегов, похожая на XML. XML используется для описания данных, а HTML — для описания отображения. HTML-теги похожи на уже упомянутое объявление `DocType` и обычно состоят из открывающих и закрывающих скобок, в которых заключается содержимое, которое может включать и другие теги.

После того как `DocType` объявлен, HTML-страница начинается с тега `html`, включающего в себя все содержимое страницы. Внутри тега `html` находятся теги `head` и `body`.

В тег `head` вы можете включить метаданные о странице, например заголовок или кодировку символов файла. Для HTML-файлов обычно используется кодировка UTF-8, но можно применить и другие. Вы также можете включить в тег `head` теги `link`, чтобы добавить ссылки на файлы, например, для стилей, с помощью которых будет отображаться содержимое страницы.

В теге `body` размещается все содержимое страницы. Как и тег `head`, он может содержать ссылки на файлы.

Теги `Script` используются, чтобы включить код на JavaScript с помощью атрибута `src`, который сообщает браузеру, где найти файл с кодом. Это еще находится в разработке, но создатели браузеров хотят, чтобы модули WebAssembly также можно было включать в веб-страницу путем размещения на странице тега `script`, похожего на `<script type="module">`.

Теги `Script` могут размещаться и в теге `head`, и в теге `body`, но до недавнего времени лучшим решением считалось размещать их в конце тега `body`. Это было связано с тем, что браузер останавливал конструирование DOM, пока скрипт не загрузится, и веб-страница реагирует лучше, если на ней что-то отображается до паузы, а не просто белый экран в начале загрузки. Сейчас в теге `Script` можно размещать атрибут `async`, сообщающий браузеру, что нужно продолжать сборку DOM и одновременно загружать файл скрипта.

ИНФОБОКС

На данной веб-странице более подробно объясняется, почему теги script было рекомендовано размещать в конце тега body: Илья Григорик, «Дополнительная интерактивность с JavaScript», Google Developers, http://mng.bz/xld7.

Браузеру не нужны пробелы в HTML-файле. Отступы и разбиения на строки в таком файле необязательны и включаются для лучшей читаемости.

Создание HTML-страницы

Следующий код HTML (листинг 3.3) — это базовая веб-страница для вашего файла WebAssembly, которую нужно разместить в папке WebAssembly\Chapter 3\3.5

js_plumbing\ и назвать `js_plumbing.html`. Веб-страница в этом листинге включает в себя только ссылку на файл JavaScript, сгенерированный Emscripten. Данный файл отвечает за загрузку и создание экземпляра вашего модуля WebAssembly, поэтому вам нужно только включить ссылку на этот файл.

Листинг 3.3. Код HTML для файла js_plumbing.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    HTML page I created for my WebAssembly module.

    <script src="js_plumbing.js"></script>
  </body>
</html>
```

Файл JavaScript отвечает за загрузку и создание экземпляра вашего модуля WebAssembly

Просмотр вашей HTML-страницы

Если открыть браузер и ввести в адресную строку `http://localhost:8080/js_plumbing.html`, то вы должны увидеть страницу, похожую на ту, что отображается на рис. 3.15.

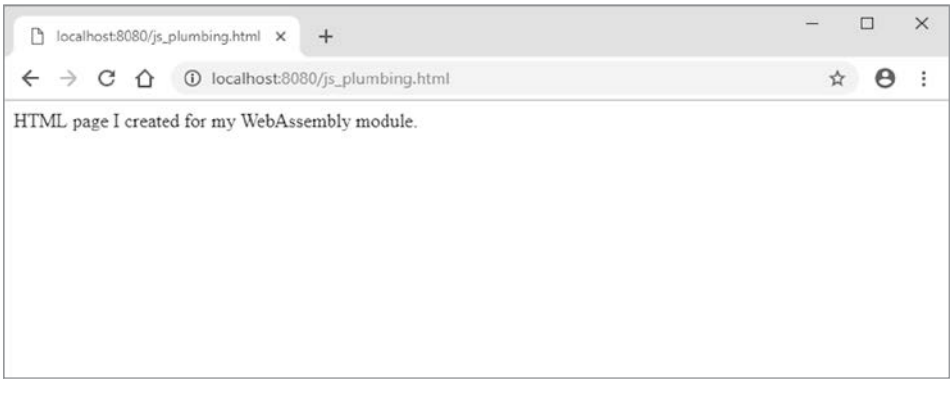


Рис. 3.15. HTML-страница, которую вы создали, открытая в Google Chrome

Посмотрев на эту веб-страницу в браузере, вы можете задаться вопросом: а где же текст, показывающий все простые числа, который я видел, когда использовал подход с шаблоном HTML в разделе 3.4?

Когда вы дали Emscripten команду сгенерировать HTML-шаблон в разделе 3.4, он разместил весь вывод `printf` в текстовом окне на веб-странице. Но по умолчанию

он перенаправляет весь вывод в консоль в инструментах разработчика в браузере. Чтобы открыть эти инструменты, нажмите F12.

Инструменты разработчика различаются в разных браузерах, но во всех есть возможность просмотреть вывод на консоль. Как можно увидеть на рис. 3.16, текст из вызова `printf` в вашем модуле выводится в окно консоли в инструментах разработчика в браузере.

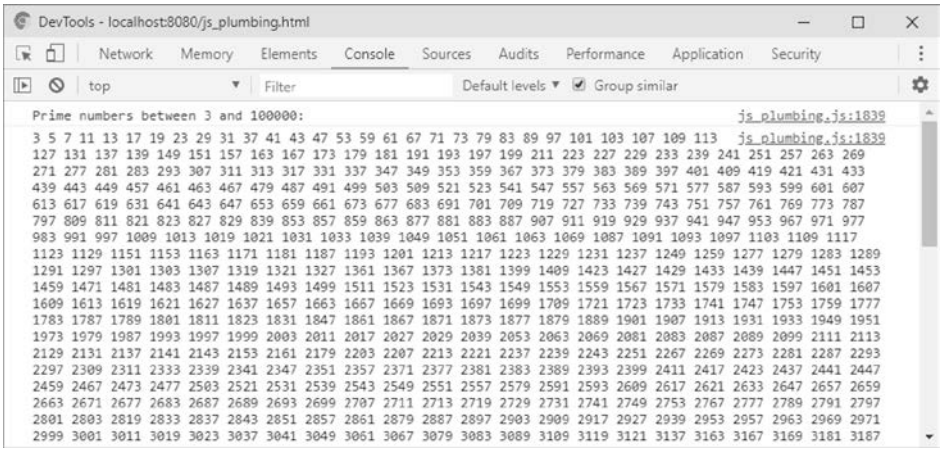


Рис. 3.16. Окно консоли в инструментах разработчика в Google Chrome, отображающее список простых чисел

3.6. EMSCRIPTEN ГЕНЕРИРУЕТ ТОЛЬКО ФАЙЛ WEBASSEMBLY

На рис. 3.17 показан третий сценарий, который мы обсудим при создании модуля WebAssembly с помощью Emscripten. В нем вы даете Emscripten команду лишь скомпилировать ваш код на C или C++ в WebAssembly и не генерировать никаких других файлов. В этом случае вам нужно не только создать HTML-файл, но и написать код на JavaScript, необходимый для загрузки и создания экземпляра модуля.

Вы можете создать модуль WebAssembly таким образом, сообщив Emscripten, что вы хотите создать вспомогательный модуль. Последний обычно используется при динамическом связывании, когда несколько модулей можно загрузить и затем связать между собой в среде выполнения, чтобы они работали как одна единица. Это похоже на использование библиотек в других языках. Мы обсудим динамическое связывание в дальнейших главах. В данном сценарии мы создаем вспомогательный модуль не для него. Вы скомандуете Emscripten создать вспомогательный модуль, поскольку в этом случае он не включает в модуль WebAssembly

ни одну функцию стандартной библиотеки C с помощью вашего кода и не создает связующий файл JavaScript.

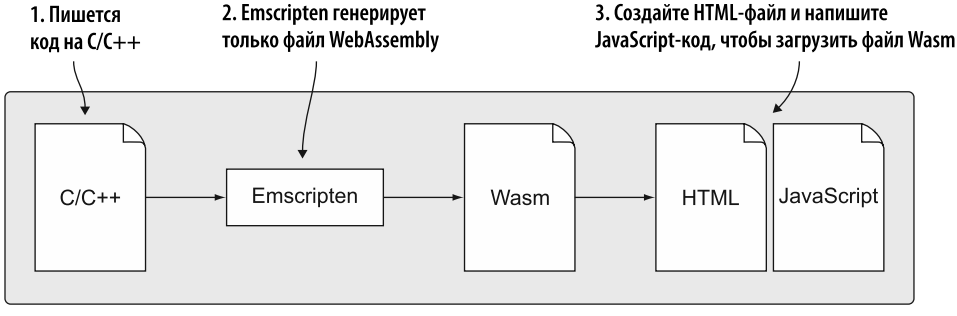


Рис. 3.17. Emscripten получил запрос о генерации только файла WebAssembly. Затем вы создаете необходимый код HTML и JavaScript, чтобы загрузить и создать экземпляр модуля

Вам может понадобиться вспомогательный модуль по нескольким причинам.

- Вы хотите применить динамическое связывание, при котором несколько модулей будут загружены и связаны воедино во время выполнения. В этом случае один из ваших модулей компилируется в качестве основного и будет содержать функции стандартной библиотеки C. Я объясню разницу между основными и вспомогательными модулями в главе 7, когда вы будете изучать динамическое связывание, но и те и другие модули подпадают под три сценария, которые мы рассматриваем в этой главе.
- Код в вашем модуле не нуждается в стандартной библиотеке C. Здесь требуется осторожность, поскольку если вы передаете между кодом на JavaScript и модулем что-то, помимо целых чисел или чисел с плавающей запятой, то необходимо управление памятью, а это потребует в каком-то виде функций стандартной библиотеки C `malloc` и `free`. Проблемы с управлением памятью повлияют только на ваш модуль, учитывая, что его память — это лишь буфер на массиве, переданный ему из JavaScript, но могут появиться ошибки, которые будет сложно отследить.
- Вы хотите узнать, как загружать, компилировать и создавать экземпляр модуля с помощью браузера, — это полезный навык, учитывая, что Emscripten не является единственным компилятором, создающим модули WebAssembly. Некоторые примеры в Интернете показывают загрузку модулей вручную, поэтому уметь создавать модули для такой загрузки полезно, если вы хотите использовать эти примеры. Есть также вероятность, что в какой-то момент вы заходите поработать со сторонним модулем, у которого нет связующего файла JavaScript.

3.6.1. Компиляция C или C++ в виде вспомогательного модуля с помощью Emscripten

Как показано на рис. 3.18, вашим первым шагом будет создание кода на C. Создайте папку, в которой будете хранить файлы для этого подраздела: WebAssembly\Chapter 3\3.6 side_module\.

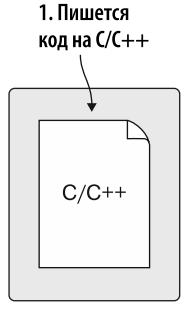


Рис. 3.18. Шаг 1 — создание кода на C или C++

Поскольку у вашего кода на C не будет доступа к функции `printf`, вам понадобится простой файл на C в качестве замены используемым до этого момента примерам. Вы создадите функцию `Increment`, которая получает целое число, прибавляет к полученному значению 1 и затем возвращает результат вызывающему. В этом случае вызывающим будет функция JavaScript. Разместите следующий код в файле `side_module.c`:

```
int Increment(int value) {
  return (value + 1);
}
```

Теперь у вас есть код на C, и вы можете сделать следующий шаг — дать Emscripten команду сгенерировать только файл WebAssembly, как показано на рис. 3.19. Чтобы скомпилировать код в виде вспомогательного модуля, нужно включить флаг `-s SIDE_MODULE=2` как часть командной строки `emcc`. Этот флаг сообщает Emscripten, что вы не хотите включать в модуль такие элементы, как функции стандартной библиотеки C, или генерировать связующий файл JavaScript.

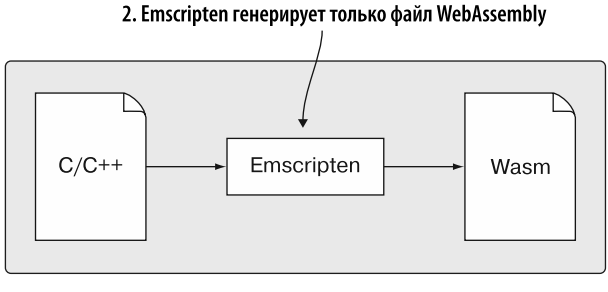


Рис. 3.19. Emscripten генерирует только файл WebAssembly

Вам также понадобится включить флаг оптимизации `-O1` (заглавная «O» и цифра 1). Если вы его не укажете, то Emscripten будет по умолчанию использовать `-O0` (заглавная «O» и цифра 0), указывающий на то, что оптимизации не нужны.

Отсутствие оптимизаций в данном сценарии вызовет ошибки в ссылках при попытке загрузить модуль — последний ожидает наличия нескольких функций и глобальных переменных, но в вашем коде их не будет. Добавление любого флага

оптимизации, кроме `-O0`, исправит эту проблему, убрав лишние импортирования, поэтому выбирайте следующий уровень флага оптимизации `-O1`. (Буква «O» чувствительна к регистру и должна быть заглавной.)

Вам нужно указать, что вы хотите экспортировать функцию `Increment`, чтобы код JavaScript мог ее вызвать. Указать это в компиляторе Emscripten можно, включив название функции в массив командной строки `-s EXPORTED_FUNCTIONS`. Emscripten добавляет символ подчеркивания перед функциями при генерировании файла WebAssembly, поэтому вам следует добавить данный символ при включении названия функции в экспортированный массив: `_Increment`.

СОВЕТ

В этом случае вам нужно указать только одну функцию в массиве командной строки EXPORTED_FUNCTIONS. Если вам понадобится указать несколько функций, то не ставьте пробел между запятой и следующей функцией, иначе получите ошибку компиляции. Если вы хотите поставить пробел между названиями функций, то следует заключить массив командной строки в двойные кавычки следующим образом: -s "EXPORTED_FUNCTIONS=['_Increment', '_Decrement']".

Наконец, указанный вами файл для вывода должен иметь расширение `.wasm`. В первом сценарии вы указывали файл HTML, а во втором — JavaScript. В этом случае вы указываете файл WebAssembly. Если не укажете его название, то Emscripten создаст файл `a.out.wasm`.

Вы можете скомпилировать ваш код `Increment` в модуль WebAssembly, открыв окно командной строки, перейдя к папке, в которой вы сохранили ваш файл на C, и затем запустив следующую команду:

```
emcc side_module.c -s SIDE_MODULE=2 -O1
➥ -s EXPORTED_FUNCTIONS=['_Increment'] -o side_module.wasm
```

Если вы посмотрите в папку, в которой хранится файл `side_module.c`, то теперь должны видеть только один новый файл, выделенный на рис. 3.20.

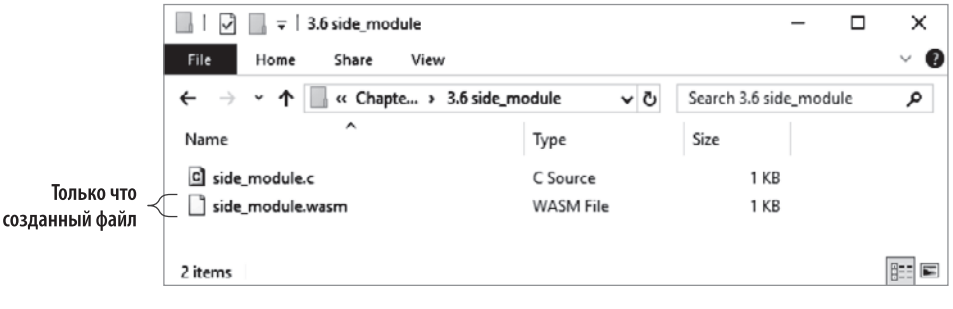


Рис. 3.20. Только что созданный файл WebAssembly

3.6.2. Загрузка и создание экземпляра модуля в браузере

Теперь, когда вы знаете, как создать сам файл Wasm, вам нужно создать HTML-файл и написать код на JavaScript, чтобы запросить этот файл с сервера и создать экземпляр модуля.

Промисы и выражения стрелочных функций

При работе со многими функциями JavaScript, которые мы сейчас обсудим, эти функции обычно работают асинхронно благодаря использованию промисов. Когда вы вызываете асинхронную функцию, она возвращает объект `Promise`, который будет вызван позже, когда действие будет либо выполнено (успешно), либо отклонено (произошла ошибка).

У объекта `Promise` есть метод `then`, получающий два параметра, которые являются функциями обратного вызова. Первый будет вызываться, если действие выполнено, а второй — если оно отклонено.

В следующем примере содержатся и функция, вызываемая при выполнении запроса, и функция, вызываемая при ошибке:

```
asyncFunctionCall.then(onFulfilled, onRejected);
```

Передает, чтобы функции обратного вызова были вызваны, когда промис либо выполнен, либо отклонен

И функция для «выполнено», и функция для «отклонено» получают один параметр. Функция, вызывающая функцию «выполнено», может передать любые данные в качестве значения параметра. Для функции «отклонено» значением параметра является строка, содержащая причину отклонения.

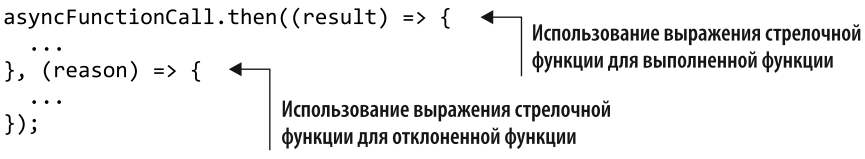
В предыдущем примере вы передали в функцию указатели, которые нужно вызвать при выполнении или отклонении метода `then`. Вместо того чтобы создавать отдельные функции где-то в других местах кода, вы всегда можете создать анонимные функции, как в примере ниже:

```
asyncFunctionCall.then(function(result) {
  ...
}, function(reason) {
  ...
});
```

Анонимная функция на случай, если промис выполнен

Анонимная функция на случай, если промис отклонен

Часто при работе с промисами можно увидеть, как эту идею развивают с помощью выражений стрелочных функций, имеющих более короткий синтаксис в сравнении с нормальными функциями, как в примере ниже:



Если есть только один параметр, то скобки необязательны. Например, функцию `(result) => {}` можно записать как `result => {}`. Если параметров нет, то скобки используются: `() => {}`.

Касательно тела выражения стрелочной функции, если ожидается возвращаемое значение и используются фигурные скобки, то требуется явный оператор `return`:

```
(value1, value2) => { return value1 + value2 }
```

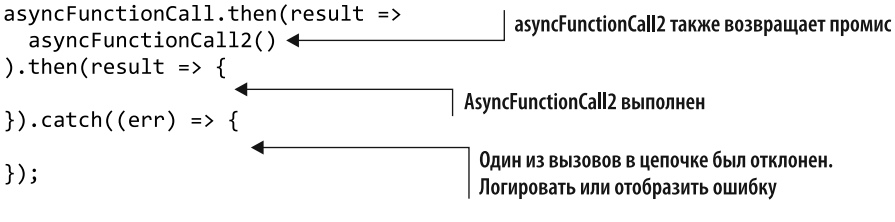
Если тело выражения стрелочной функции заключено в круглые скобки или их нет совсем, то оператор `return` неявный:

```
(value1, value2) => value1 + value2
```

Если вас интересует только информация о том, было ли выполнено действие, то вам не нужно указывать второй параметр в алгоритме `then` для отклонения.

Если, наоборот, действие интересует вас лишь в случае ошибки, то можете указать `null` в качестве первого параметра и функцию обратного вызова для отклонения. Но обычно, если вас интересует только факт ошибки, то стоит использовать алгоритм `catch`. Он получает один параметр — функцию обратного вызова, которая будет вызвана в случае отклонения действия.

И `then`, и `catch` возвращают промисы, что позволяет связать несколько асинхронных операций. Это сильно облегчает работу с несколькими взаимозависимыми асинхронными операциями, поскольку следующий алгоритм `then` будет вызван, только если предыдущий будет выполнен:



Короткий способ создания объектов JavaScript

Ряд функций, которые вы будете использовать в следующих примерах, принимают объекты в качестве параметров. Вы можете создать объект в `JavaScript` с помощью

`new Object()`, но есть и короткий способ создания объектов с использованием фигурных скобок, как в примере ниже, в котором создается пустой объект:

```
const person = {};
```

В объект можно включить пары «имя — значение», отделяя каждую пару запятой. Сама пара разделена двоеточием, а значение может быть `string`, `number`, `object`, `array`, `true`, `false` или `null`. Значения строк заключены в одинарные или двойные кавычки. Ниже дан пример пары «имя — значение»:

```
age: 21
```

Создание объектов, таким образом, упрощает работу, поскольку объект можно объявить и инициализировать за один этап. Определив объект JavaScript, можно получить доступ к характеристикам с помощью записи через точку:

```
const person = { name: "Sam Smith", age: 21 };
console.log("The person's name is: " + person.name);
```

Обзор API WebAssembly на JavaScript

Браузеры, поддерживающие WebAssembly, предоставляют API WebAssembly на JavaScript. Этот API представляет собой пространство имен WebAssembly с несколькими функциями и объектами, используемыми для компиляции и создания экземпляра модуля, для взаимодействия с некоторыми аспектами модуля, например памятью, для передачи строк между модулем и JavaScript и для обработки ошибочных условий.

При использовании файла JavaScript, сгенерированного Emscripten, он выполняет процесс загрузки файла WebAssembly за вас. Затем он взаимодействует с API WebAssembly на JavaScript, чтобы скомпилировать и создать экземпляр модуля.

Здесь вы увидите, как используется этот API, чтобы вы могли взаимодействовать с ним для загрузки вручную модуля WebAssembly, созданного вами в подразделе 3.6.1.

СПРАВКА

Многие современные «настольные» браузеры и браузеры для мобильных устройств, включая Edge, Firefox, Chrome, Safari и Opera, поддерживают WebAssembly. Подробный список можно просмотреть на сайте https://caniuse.com/#search=WebAssembly.

Прежде чем вы сможете что-то сделать с модулем WebAssembly, нужно сначала дать команду на загрузку файла WebAssembly. Чтобы запросить файл, используйте метод JavaScript `fetch`. Он позволяет JavaScript делать вызовы, связанные с HTTP, асинхронно. Если вам нужно лишь получить данные, а не загрузить их на сервер, например, то следует определить только первый параметр, то есть URI файла,

который вы хотите загрузить, и алгоритм `fetch` вернет объект `Promise`. Например, если файл Wasm находится в той же папке на сервере, откуда был загружен HTML-файл, то вам понадобится только указать в качестве URI название файла:

```
fetch("side_module.wasm")
```

Метод `fetch` получает объект JavaScript в качестве необязательного второго параметра для контроля множества настроек, связанных с запросом, например типа контента данных, если вы передаете данные на сервер. В этой книге мы не будем использовать необязательный второй параметр, называемый `init`, но если вам нужно знать подробности о нем, то они доступны на сайте MDN Web Docs: http://mng.bz/ANle.

После того как будет получен файл WebAssembly, вам нужно будет как-то скомпилировать и создать экземпляр модуля. Рекомендованным способом является функция `WebAssembly.instantiateStreaming`, поскольку модуль компилируется в машинный код в процессе загрузки байт-кода методом `fetch`. Такая компиляция сокращает время начала работы, ввиду того что модуль готов к созданию экземпляра сразу по окончании загрузки.

Функция `instantiateStreaming` получает два параметра. Первый — это объект `Response` или объект `Promise`, который выполнится с помощью объекта `Response`, представляющие источник в виде файла Wasm. Метод `fetch` возвращает объект `Response`, поэтому вы можете просто включить вызов алгоритма в качестве первого параметра `instantiateStreaming`. Второй параметр — это необязательный объект JavaScript (его мы кратко обсудим), в котором вы передаете модулю ожидаемые им данные, например импортированные функции или глобальные переменные.

Функция `instantiateStreaming` возвращает объект `Promise`, который, будучи выполненным, станет содержать свойства `module` и `instance`. Свойство `module` — это объект `WebAssembly.Module`, а свойство `instance` — объект `WebAssembly.Instance`, тот самый, который нас интересует, поскольку он содержит свойство `exports`, хранящее все объекты, которые модуль экспортирует.

Ниже представлен пример использования функции `WebAssembly.instantiate Streaming` для загрузки созданного вами в подразделе 3.6.1 модуля:

Объект Promise из запроса fetch передается в качестве первого параметра

В объекте instance вы можете получить доступ к экспортированной функции

```
WebAssembly.instantiateStreaming(fetch("side_module.wasm"),
➥ importObject).then(result => {
  const value = result.instance.exports._Increment(17);
  console.log(value.toString());
});
```

Функция `instantiateStreaming` была добавлена в браузеры после первого релиза MVP WebAssembly, поэтому возможно, что некоторые браузеры, поддерживающие WebAssembly, не поддерживают данную функцию. Лучше использовать проверку поддержки, чтобы проверить доступность функции, прежде чем применять ее. В конце этой главы, в разделе 3.7, я покажу, как проверить доступность. Если функция недоступна, то нужно использовать более старую функцию `WebAssembly.instantiate`.

СОВЕТ

На сайте MDN Web Docs (ранее — Mozilla Developer Network) есть статья о функции instantiateStreaming плюс актуальная таблица совместимости браузеров внизу страницы: http://mng.bz/ZeoN.

Как и при вызове `instantiateStreaming`, с функцией `instantiate` вы также можете использовать `fetch` для загрузки содержимого файла WebAssembly. Но в отличие от `instantiateStreaming` вы не можете передать объект `Promise` напрямую функции `instantiate`. Вместо этого нужно подождать, пока запрос `fetch` будет выполнен, конвертировать данные в `ArrayBuffer` и затем передать последний функции `instantiate`. Эта функция также получает необязательный второй параметр в виде объекта JavaScript для импортированных функций модуля.

Ниже представлен пример использования функции `WebAssembly.instantiate`:

```
fetch("side_module.wasm").then(response =>
  response.arrayBuffer()
).then(bytes =>
  WebAssembly.instantiate(bytes, importObject)
).then(result => {

  const value = result.instance.exports._Increment(17);
  console.log(value.toString());
});
```

Отдает команду на загрузку файла WebAssembly

Отдает команду на превращение данных из файла в ArrayBuffer

Передает ArrayBuffer функции instantiate

Теперь у вас есть доступ к экземпляру модуля: result.instance

В главе 9 вы будете работать просто со скомпилированным модулем (без созданного экземпляра), передавая его из веб-воркера. В тот момент вы также будете работать с функциями `WebAssembly.compileStreaming` и `WebAssembly.compile`. Сейчас скажу только, что функции `compileStreaming` и `compile` работают так же, как `instantiateStreaming` и `instantiate`, но возвращают лишь скомпилированный модуль.

Обратите внимание: существует функция `WebAssembly.Module`, которая может скомпилировать модуль, и функция `WebAssembly.Instance`, которая создает экземпляр скомпилированного модуля, но я не рекомендую их использовать, поскольку их вызовы синхронны. Функции `instantiateStreaming`, `instantiate`, `compileStreaming` и `compile` — асинхронны, рекомендую применять их.

Как я уже упоминал, необязательный объект JavaScript (часто называемый `importObject`) может передаваться в качестве второго параметра функциям `instantiateStreaming` и `instantiate`, чтобы предоставить модулю все, что тому нужно импортировать. Данный объект может содержать память, таблицу, глобальные переменные или ссылки на функции. Вы будете работать с этими импортируемыми объектами, изучая разные примеры ниже.

Модули WebAssembly могут содержать раздел «Память», указывающий, сколько страниц памяти должно быть в нем изначально и дополнительно максимальное количество страниц. Каждая страница памяти содержит 65 536 байт, или 64 Кбайт. Если модуль указывает, что необходимо импортировать память, то ваш код на JavaScript должен предоставить ее в качестве части `importObject`, который передается функции `instantiateStreaming` или `instantiate`.

ИНФОБОКС

Одна из характеристик безопасности WebAssembly заключается в том, что модуль не может размещать собственную память или менять ее размер напрямую. Вместо этого память, используемая модулями WebAssembly, предоставляется хостом в форме ArrayBuffer с изменяемым размером при создании экземпляра модуля.

Чтобы передать память в модуль, первым делом нужно создать экземпляр объекта `WebAssembly.Memory` в качестве части `importObject`. Этот объект часто получает объект JavaScript в качестве части своего конструктора. Первое свойство объекта JavaScript — `initial`, указывающее на то, сколько страниц памяти должно быть изначально выделено для данного модуля. Объект JavaScript дополнительно может включать и свойство `maximum`, указывающее на максимальное количество страниц, до которого можно увеличивать память WebAssembly. Вы узнаете более подробно об увеличении памяти позже.

Ниже представлен пример создания объекта `WebAssembly.Memory` и передачи его в модуль:

```
const importObject = {
  env: {
    memory: new WebAssembly.Memory({initial: 1, maximum: 10})
  }
};

WebAssembly.instantiateStreaming(fetch("test.wasm"),
➥ importObject).then(result => { ... });
```

Изначально — одна страница памяти. Можно увеличить максимально до десяти страниц

Создание JavaScript для загрузки и создания экземпляра модуля

Вы напишете код на JavaScript, чтобы загрузить файл `side_module.wasm`, который создали в подразделе 3.6.1, и используете функцию `WebAssembly.instantiateStreaming`. В подразделе 3.6.1 вы дали Emscripten команду создать модуль

как вспомогательный, чтобы он не включал в файл Wasm функции стандартной библиотеки C и не создавал связующий файл JavaScript. Хотя мы не собирались в этой главе использовать его по прямому назначению, поскольку вспомогательные модули создаются в Emscripten для динамического связывания двух модулей в среде выполнения, Emscripten добавляет в модуль импортированные объекты, которые нужно будет предоставить при вызове `instantiateStreaming`.

Вам нужно будет определить объект JavaScript, назвав его `importObject`, у которого есть вложенный объект `env`, в свою очередь содержащий свойство `__memory_base`, которое хочет импортировать модуль. Оно будет просто содержать нулевое значение, поскольку вы не станете динамически связывать этот модуль.

Создав `importObject`, вы можете вызвать функцию `instantiateStreaming`, передав ей результат метода `fetch` для файла Wasm в качестве первого параметра и `importObject` в качестве второго. Функция `instantiateStreaming` возвращает промис, поэтому вы установите обработчик для функции обратного вызова, который будет вызван после успешной загрузки, компиляции и создания экземпляра модуля. В этот момент вы можете получить доступ к экспортированным элементам экземпляра модуля WebAssembly и вызвать свою функцию `_Increment`, передав ей значение 17. Ваша функция `_Increment` примет переданное значение, добавит к нему 1 и вернет новое значение. Вызов `console.log`, который вы добавите, выведет результат в окно консоли браузера и отобразит в этом случае число 18.

В листинге 3.4 представлен код на JavaScript, необходимый для загрузки и создания экземпляра вашего модуля.

Листинг 3.4. Код JavaScript, необходимый для загрузки и создания экземпляра side_module.wasm

```
const importObject = {
  env: {
    __memory_base: 0,
  }
};

WebAssembly.instantiateStreaming(fetch("side_module.wasm"),
➥ importObject).then(result => {
  const value = result.instance.exports._Increment(17);
  console.log(value.toString());
});
```

Создание базовой HTML-страницы

В папке Chapter 3\3.6 side_module\ создайте файл `side_module.html`, затем откройте его в своем любимом редакторе. Как можно увидеть в листинге 3.5, HTML, который вы собираетесь использовать для загрузки файла WebAssembly, почти идентичен тому, который вы задействовали в файле `js_plumbing.html` в подразделе 3.5.2, за исключением того, что здесь вы будете не ссылаться на файл JavaScript, а возьмете код на JavaScript, написанный в листинге 3.4, и добавите его в блок скриптов в листинге 3.5.

Листинг 3.5. HTML-страница для модуля WebAssembly side_module.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    HTML page I created for my WebAssembly module.

    <script>
      const importObject = {
        env: {
          __memory_base: 0,
        }
      };

      WebAssembly.instantiateStreaming(fetch("side_module.wasm"),
      ➥ importObject).then(result => {
        const value = result.instance.exports._Increment(17);
        console.log(value.toString());
      });
    </script>
  </body>
</html>
```

Откройте браузер и введите в адресную строку `http://localhost:8080/side_module.html`. Затем нажмите F12, чтобы открыть инструменты разработчика в браузере, и увидите, что созданная вами HTML-страница отображает число 18, как показано на рис. 3.21.

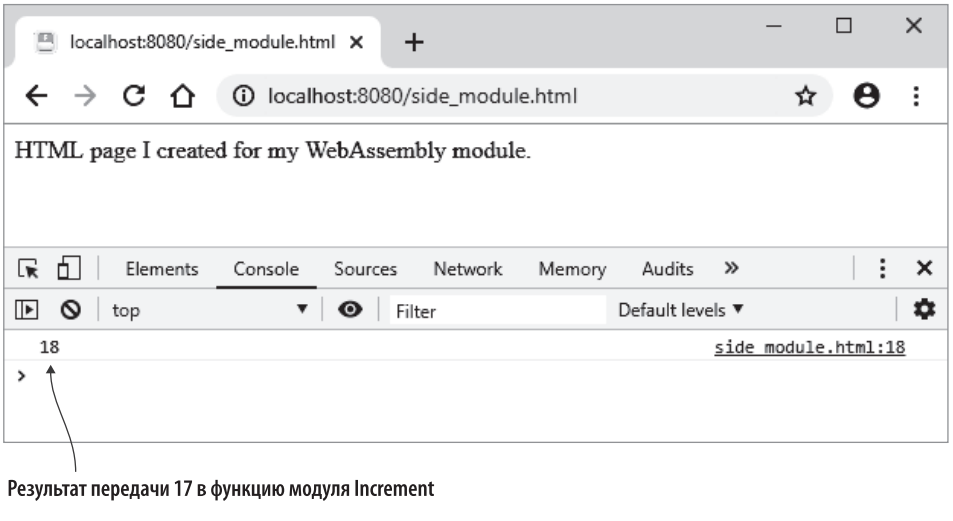


Рис. 3.21. Созданная вами HTML-страница, показывающая результат вызова функции Increment

3.7. ПРОВЕРКА ПОДДЕРЖКИ: КАК ПРОВЕРИТЬ, ДОСТУПЕН ЛИ WEBASSEMBLY

С развитием новых технологий некоторые производители браузеров иногда реализуют какую-нибудь функцию раньше других. Кроме того, не все обновляют свои браузеры до последней версии так часто, как нам бы хотелось. Поэтому, даже если пользователь работает с браузером от производителя, реализовавшего эту функцию, он может не применять версию, поддерживающую ее. Если существует вероятность того, что ваши пользователи будут задействовать браузер, в котором нет нужной вам функции, то лучше всего будет проверить ее наличие, прежде чем применять ее.

WebAssembly — достаточно новая технология, и не все используемые сейчас браузеры — и версии Node.js — поддерживают его. Вдобавок может быть, что браузер поддерживает WebAssembly, но не дает загрузить и создать экземпляр запрошенного вами модуля из-за проверок безопасности, например *политики защиты контента* (Content Security Policy, CSP) — дополнительного уровня безопасности, который пытается запретить такие вещи, как *межсайтовый скриптинг* (cross-site scripting, XSS) и атаки с помощью внедрения данных. Из-за этого недостаточно просто проверить, существует ли объект WebAssembly на JavaScript. Чтобы определить, поддерживает ли браузер или Node.js WebAssembly, можно использовать функцию, показанную в листинге 3.6.

Листинг 3.6. JavaScript для проверки того, поддерживается ли WebAssembly

Обернуто в try-catch на случай возникновения CompileError или LinkError

Проверяет, существует ли объект API WebAssembly на JavaScript

Компилирует минимальный модуль только с волшебным числом ('\0asm') и номером версии (1)

```
function isWebAssemblySupported() {
  try {
    if (typeof WebAssembly === "object") {
      const module = new WebAssembly.Module(new Uint8Array([0x00,
    ➥ 0x61, 0x73, 0x6D, 0x01, 0x00, 0x00, 0x00]));
      if (module instanceof WebAssembly.Module) {
        const moduleInstance = new WebAssembly.Instance(module);
        return (moduleInstance instanceof WebAssembly.Instance);
      }
    }
  } catch (err) {}

return false;
}

console.log((isWebAssemblySupported() ? "WebAssembly поддерживается":
    "WebAssembly не поддерживается"));
```

Поддерживает WebAssembly, если объект является API WebAssembly.Instance на JavaScript

Если результат является объектом API WebAssembly.Module на JavaScript

WebAssembly не поддерживается

Проверяет, является ли результат объектом API WebAssembly.Module на JavaScript

Теперь вы знаете, как проверить, поддерживается ли WebAssembly. Но остается вероятность того, что браузер или Node.js не будут поддерживать последнюю функцию. Например, `WebAssembly.instantiateStreaming` является новой функцией JavaScript, которую можно использовать вместо функции `WebAssembly.instantiate`, но `instantiateStreaming` была создана после релиза MVP. В итоге функция `instantiateStreaming` может существовать не во всех браузерах, поддерживающих WebAssembly. Чтобы проверить, есть ли функция JavaScript, можно сделать следующее:

```
if (typeof WebAssembly.instantiateStreaming === "function") {
  console.log("You can use the WebAssembly.instantiateStreaming
➥ function");
} else {
  console.log("The WebAssembly.instantiateStreaming function is not
➥ available. You need to use WebAssembly.instantiate instead.");
}
```

В подобных проверках вы сначала проверяете наличие функции, которую хотели бы использовать, а затем переходите к альтернативам, если она недоступна. В нашем случае предпочтительна функция `instantiateStreaming`, поскольку она компилирует код в процессе загрузки модуля, но в случае ее недоступности функция `instantiate` тоже сойдет. У нее просто нет того улучшения производительности, которое привносит `instantiateStreaming`.

А теперь спросим, как же то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже представлены некоторые возможные варианты использования того, что вы узнали в этой главе.

- Вы можете использовать параметр вывода в шаблоне HTML Emscripten, чтобы быстро создать код для прототипирования или для того чтобы проверить какую-либо возможность WebAssembly отдельно от вашей веб-страницы. С помощью функции `printf` можно вывести информацию в текстовое поле на веб-странице и консоль инструментов разработчика в браузере, чтобы подтвердить, что все работает как должно. Если ваш код работает в тестовом окружении, то можете применить его и в основной базе кода.
- Вы можете использовать API WebAssembly на JavaScript для проверки, поддерживается ли WebAssembly.
- Другие примеры включают в себя калькулятор или конвертер единиц (например, градусов по Цельсию в градусы по Фаренгейту или сантиметров в дюймы).

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какие четыре типа данных поддерживает WebAssembly?
2. Добавьте функцию `Decrement` во вспомогательный модуль, созданный в подразделе 3.6.1:
   а) функция должна иметь целочисленное возвращаемое значение и целочисленный параметр. Вычтите 1 из полученного значения и верните результат вызывающей функции;
   б) скомпилируйте вспомогательный модуль, затем отредактируйте JavaScript, чтобы вызвать функцию и отобразить результат в консоли.

РЕЗЮМЕ

Как вы увидели в этой главе, набор инструментов Emscripten использует инструментарий компилятора LLVM, чтобы конвертировать код на C или C++ в IR LLVM. Затем Emscripten конвертирует IR LLVM в байт-код WebAssembly. Браузеры, поддерживающие WebAssembly, загружают файл WebAssembly, и если все работает правильно, то байт-код компилируется дальше в машинный код устройства.

Набор инструментов Emscripten подстраивается под ваши потребности, позволяя создавать модули несколькими разными способами.

- Вы можете создать модуль и дать Emscripten команду сгенерировать за вас файлы HTML и JavaScript. Этот подход полезен, если вы хотите обучиться созданию модулей WebAssembly, прежде чем изучать необходимые функции HTML и JavaScript. Еще он полезен, если вам нужно быстро что-то проверить, не создавая файлы HTML и JavaScript.
- Вы можете создать модуль и дать Emscripten команду сгенерировать за вас файл JavaScript. Здесь вы отвечаете за создание собственного HTML-файла. Благодаря этому у вас появляется выбор: создать новую страницу HTML или просто добавить ссылку на сгенерированный файл JavaScript на существующую веб-страницу. Данный метод обычно используется для производственного кода.
- Наконец, вы можете создать только сам модуль. Здесь вы отвечаете и за создание собственного HTML-файла, и за JavaScript, необходимый для загрузки и создания экземпляра модуля. Этот подход может пригодиться для изучения деталей API WebAssembly на JavaScript.

Часть II

Работа с модулями

Теперь, когда вы знаете, что такое WebAssembly, и уже знакомы с набором инструментальных средств Emscripten, эта часть книги научит вас создавать модули WebAssembly, с которыми может взаимодействовать JavaScript-код (и наоборот).

В главе 4 вы узнаете, как взять существующую кодовую базу на C или C++ и настроить так, чтобы ее можно было скомпилировать в модуль WebAssembly. Вы узнаете, как взаимодействовать с новым модулем с помощью JavaScript-кода веб-страницы.

В главе 5 рассказывается, как настроить код из главы 4 так, чтобы модуль WebAssembly мог вызывать JavaScript-код вашей веб-страницы.

Глава 6 переводит вызов JavaScript-кода вашей веб-страницы на другой уровень — вы научитесь передавать указатели функций JavaScript в модуль WebAssembly. Это позволяет JavaScript определять функции по требованию и использовать промисы JavaScript.

4 Повторное использование существующей кодовой базы на C++

В этой главе

- ✓ Адаптация кода на C++ для компиляции с применением Emscripten.
- ✓ Экспорт функций WebAssembly для вызова их с помощью JavaScript.
- ✓ Вызов функций WebAssembly с использованием вспомогательных функций Emscripten.
- ✓ Передача строк и массивов в модуль WebAssembly через память модуля.

Обычно, когда речь идет о преимуществах WebAssembly, имеется в виду улучшение производительности. Однако есть еще одно значимое преимущество WebAssembly — повторное использование кода. Вместо того чтобы создавать логику несколько раз для каждой целевой среды (приложение для компьютера, сайт и др.), WebAssembly позволяет повторно задействовать один и тот же код в нескольких местах.

Представьте, что компании, у которой уже есть приложение для продаж для ПК, написанное на C++, требуется веб-версия программы. Первое, что нужно сделать, — создать веб-страницу Edit Product (Изменить товар), показанную на рис. 4.1. Новый сайт также должен использовать Node.js для серверной части, но к обсуждению Node.js мы перейдем в одной из следующих глав.

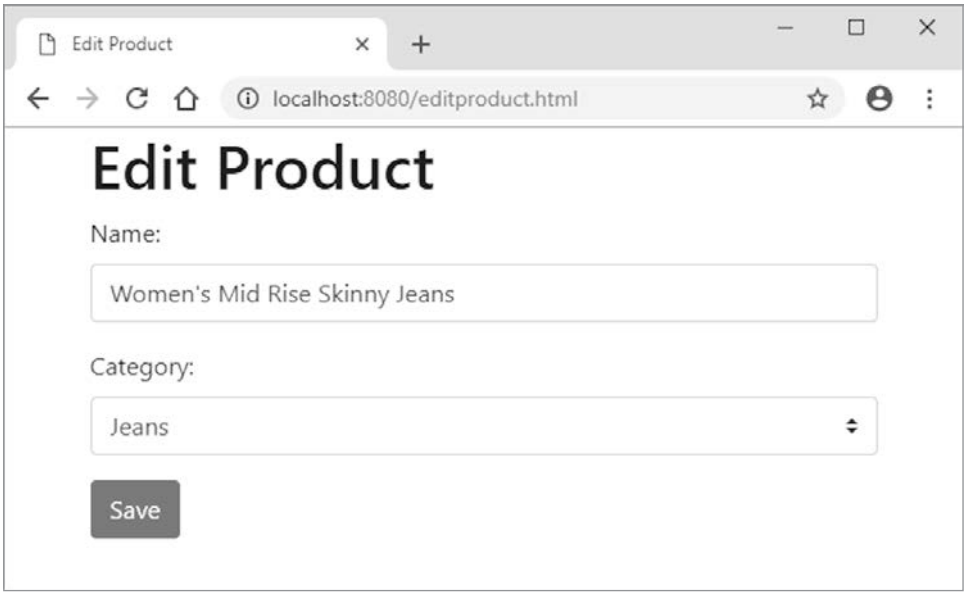


Рис. 4.1. Страница Edit Product (Изменить товар), которую нужно разработать

Поскольку у нас уже есть код на C++, стоит использовать преимущество WebAssembly — расширить код валидации для браузера и Node.js. Это позволит убедиться, что все три месторасположения проверяют данные одинаково и используют одну и ту же кодовую базу, тем самым упрощая сопровождение. Как показано на рис. 4.2, шаги по созданию сайта и встраиванию логики валидации включают следующие действия.

1. Изменить код на C++, чтобы его можно было скомпилировать с помощью Emscripten.
2. Воспользовавшись Emscripten, сгенерировать связующие файлы WebAssembly и JavaScript.
3. Создать веб-страницу и написать необходимый JavaScript-код для взаимодействия с модулем WebAssembly.

Зачем проверять пользовательский ввод дважды? Почему бы не пропустить проверку в браузере и просто проверить данные на сервере? Данные в браузере, помимо данных на сервере, проверяются по нескольким причинам.

- В основном пользователь может физически не находиться рядом с сервером. Чем дальше он находится, тем больше времени требуется на отправку данных на сервер и получение ответа. Если пользователь находится на другом конце

света, то эта задержка заметна, поэтому валидация в браузере делает сайт более отзывчивым для пользователя.

- Перенос большей части проверок в браузер также сокращает объем работы, которую необходимо выполнить серверу. Если серверу не требуется часто возвращать ответы для каждого пользователя, то он может обрабатывать больше пользователей одновременно.

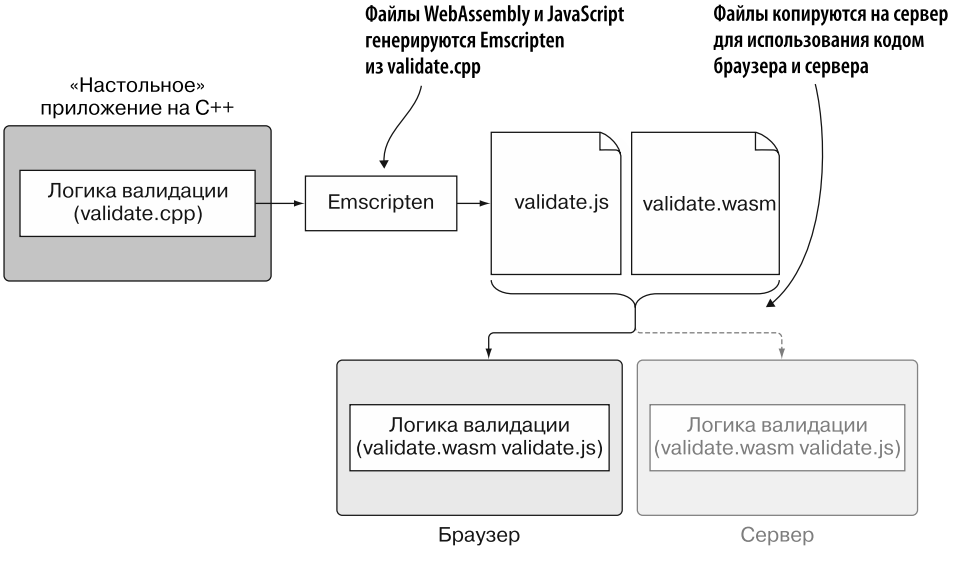


Рис. 4.2. Шаги для переноса существующего кода на C++ в модуль WebAssembly для использования на стороне браузера и сервера. Серверная часть, Node.js, обсуждается в одной из следующих глав

Какой бы нужной ни была валидация пользовательских данных в браузере, не стоит предполагать, что на сервер отправляются идеальные данные; есть способы обойти браузерные проверки. Нельзя рисковать добавлением неверных данных в БД — отправленных пользователем случайно или намеренно. Независимо от того, насколько хороша валидация в браузере, код на стороне сервера всегда должен проверять данные, которые он получает.

На рис. 4.3 показано, как проверка будет работать на веб-странице, которую вы собираетесь создать. Когда пользователь вводит некую информацию, а затем нажимает кнопку Save (Сохранить), эта информация проверяется, чтобы убедиться, что данные соответствуют ожиданиям. Если данные не пройдут проверку, то на веб-странице отобразится сообщение об ошибке. Исправив проблему, пользователь может снова нажать кнопку Save (Сохранить). Корректные данные будут отправлены на сервер.

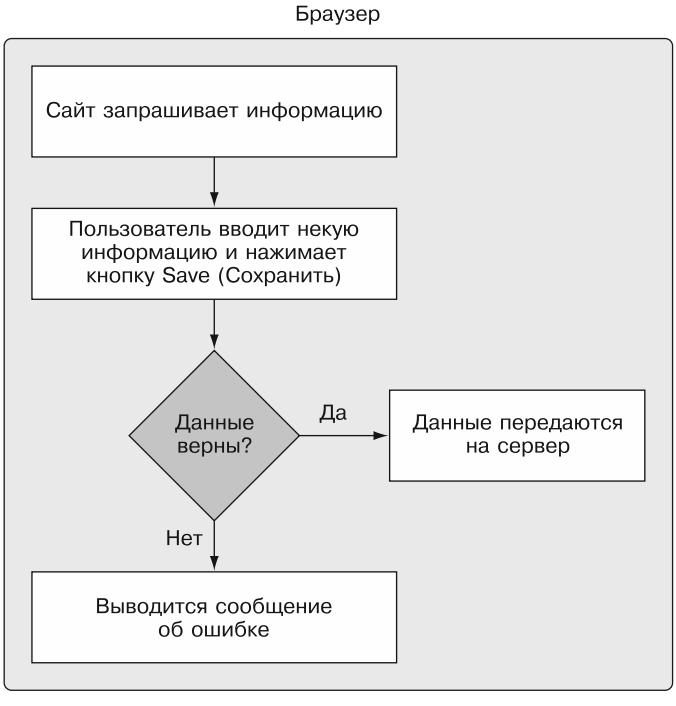


Рис. 4.3. Как работает валидация в браузере

4.1. ИСПОЛЬЗОВАНИЕ C ИЛИ C++ ДЛЯ СОЗДАНИЯ МОДУЛЯ СО СВЯЗУЮЩИМ КОДОМ EMSCRIPTEN

В текущем разделе вы узнаете, как создать код на C++ для валидации данных; вам понадобятся стандартная библиотека C и вспомогательные функции Emscripten — этот способ рекомендуется для создания модуля, который можно использовать для реальных задач. Ниже представлены причины, почему стоит придерживаться этого способа.

- Emscripten предоставляет большое количество вспомогательных функций, которые упрощают взаимодействие модуля и JavaScript.
- Emscripten также включает в модуль функции из стандартной библиотеки C в случае их использования в исходном коде. Если функция из стандартной библиотеки C нужна при выполнении, но не при компиляции, то ее можно добавить с помощью параметра командной строки.
- Если между модулем и JavaScript нужно передать что-то помимо целых чисел или чисел с плавающей запятой, то понадобится линейная память модуля.

Стандартная библиотека C включает функции `malloc` и `free` для управления памятью.

Далее в этой главе мы рассмотрим пример создания модуля WebAssembly, который не включает стандартную библиотеку C или вспомогательные функции Emscripten.

4.1.1. Внесение изменений в код на C++

Первое, что нужно сделать, — создать папку для хранения файлов этого подраздела: WebAssembly\Chapter 4\4.1 js_plumbing\source\.

Как показано на рис. 4.4, чтобы создать сайт, повторно использующий код валидации на C++, первым делом необходимо изменить код, это позволит компилировать его с помощью Emscripten.

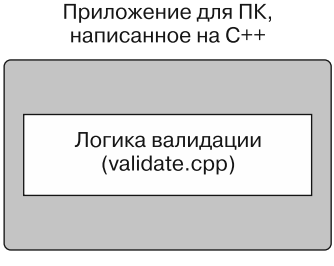


Рис. 4.4. Повторное использование кода на C++ первым делом требует изменения кода так, чтобы его можно было компилировать с помощью Emscripten

Символ условной компиляции Emscripten и заголовочный файл

Зачастую, когда при создании модуля WebAssembly вы используете уже применяющийся в программе код на C или C++, нужно будет добавить еще какие-то компоненты, чтобы заставить все это работать. Например, когда код компилируется для «настольного» приложения, ему не нужен заголовочный файл Emscripten; потребуется найти способ добавить этот заголовочный файл, но только для случаев, когда код компилируется с помощью Emscripten.

К счастью, Emscripten объявляет символ условной компиляции `__EMSCRIPTEN__`, который можно использовать, чтобы определить, компилируется ли решение с помощью Emscripten. При необходимости вы также можете включить обратное условие с проверкой символа условной компиляции, чтобы добавить заголовочные файлы, которые нужны, когда код компилируется не с помощью Emscripten.

Создайте файл `validate.cpp` и откройте его. Добавьте заголовочные файлы для стандартной библиотеки C и строковой библиотеки. Поскольку этот код является частью существующего решения, то необходимо добавить заголовочный файл для

библиотеки Emscripten, обернув его проверкой символа условной компиляции. Это позволит убедиться, что библиотека добавляется только в том случае, если Emscripten компилирует код:

```
#include <cstdlib>
#include <cstring>

#ifdef __EMSCRIPTEN__
  #include <emscripten.h>
#endif
```

Символ объявлен, когда код компилируется с помощью Emscripten

Заголовочный файл библиотеки Emscripten

СПРАВКА

Некоторые заголовочные файлы C устарели или больше не поддерживаются в C++. Пример — stdlib.h, вместо которого теперь нужно использовать cstdlib. Полный список изменений заголовочных файлов доступен на сайте https://en.cppreference.com/w/cpp/header.

Блок extern "C"

В C++ имена функций могут быть перегружены, поэтому для того, чтобы имя было уникальным во время компиляции, компилятор изменяет его, добавляя информацию о параметрах функции. Компилятор, изменяющий имена функций при компиляции кода, может стать проблемой для внешнего кода, который хочет вызвать определенную функцию, поскольку имя данной функции перестанет существовать.

Может возникнуть необходимость дать компилятору указание не изменять имена функций, вызываемых JavaScript-кодом. Для этого нужно добавить блок `extern "C"` вокруг функций. Все функции, добавляемые в данный файл, помещаются в блок. Добавьте в файл `validate.cpp` следующее:

```
#ifdef __cplusplus
extern "C" {
#endif

#ifdef __cplusplus
}
#endif
```

Компилятор не переименует функции внутри этих фигурных скобок

Все ваши функции WebAssembly будут располагаться здесь

Функция ValidateValueProvided

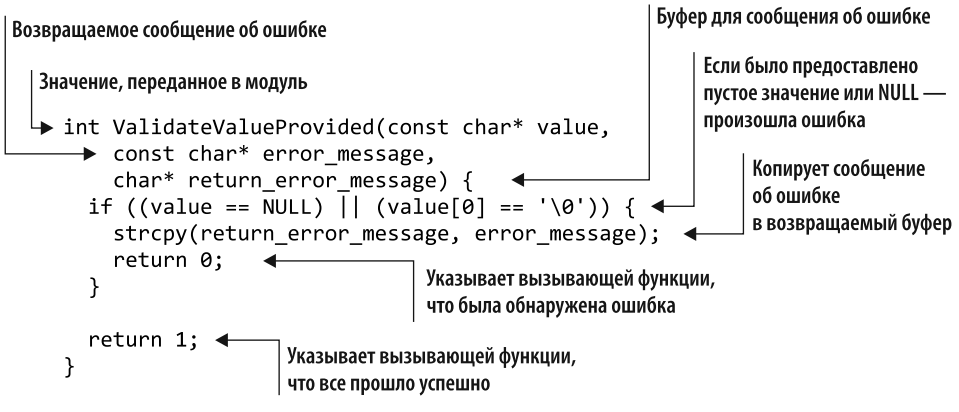
На создаваемой вами веб-странице Edit Product (Изменить товар) будет поле с названием товара и раскрывающийся список категорий, которые нужно будет проверить. И имя, и выбранная категория будут переданы в модуль в виде строк, но идентификатор категории будет содержать числовое значение.

Создадим две функции, `ValidateName` и `ValidateCategory`, для проверки названия товара и выбранной категории. Поскольку обе функции должны гарантировать,

что значение было предоставлено, добавим вспомогательную функцию с именем `ValidateValueProvided`, которая будет принимать следующие параметры:

- значение, переданное модулю веб-страницей;
- соответствующее сообщение об ошибке от модуля в зависимости от того, какая функция вызывается: `ValidateName` или `ValidateCategory`. Если значение не указано, то это сообщение будет помещено в возвращаемый буфер третьего параметра;
- буфер, в который будет помещено сообщение об ошибке, если значение не указано.

Поместите следующий код в фигурные скобки `extern "C"` файла `validate.cpp`:



Функция ValidateName

Теперь создадим функцию `ValidateName`, которая принимает следующие параметры:

- введенное пользователем название товара;
- значение максимальной длины названия;
- указатель на буфер, в который вы добавите сообщение об ошибке в случае, если будет найдена проблема при проверке.

Функция проверит два условия:

- было ли указано название товара? Можно убедиться в этом, передав название во вспомогательную функцию `ValidateValueProvided`;
- вы также убедитесь, что длина указанного названия не превышает максимальное значение длины, используя стандартную библиотечную функцию `strlen`.

Если какая-либо проверка не пройдет успешно, то поместите соответствующее сообщение об ошибке в возвращаемый буфер и завершите работу функции, вернув `0` (ошибка). Если код выполнится до конца функции, значит, проблем с проверкой не было, поэтому возвращается сообщение `1` (успех).

Добавим также объявление `EMSCRIPTEN_KEEPALIVE` в функцию `ValidateName` и обернем его проверкой символа условной компиляции с целью убедиться, что оно будет добавлено только в том случае, если код компилируется с помощью Emscripten. В главе 3 вы добавили функции из модуля к параметру командной строки Emscripten `EXPORTED_FUNCTIONS`, чтобы JavaScript-код мог взаимодействовать с этими функциями. Объявление `EMSCRIPTEN_KEEPALIVE` автоматически добавляет связанную функцию к экспортируемым, так что ее не нужно явно указывать в командной строке.

Код, показанный в листинге 4.1, — это функция `ValidateName`. Добавьте его после функции `ValidateValueProvided` в файле `validate.cpp`.

Листинг 4.1. Функция ValidateValueProvided в файле validate.cpp

```
...
#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
int ValidateName(char* name,
    int maximum_length,
    char* return_error_message) {
  if (ValidateValueProvided(name,
      "A Product Name must be provided.",
      return_error_message) == 0) {
    return 0;
  }

  if (strlen(name) > maximum_length) {
    strcpy(return_error_message, "The Product Name is too long.");
    return 0;
  }

  return 1;
}
```

Добавление функции в список экспортируемых функций

Название товара, передаваемое в модуль

Максимально допустимая длина для названия

Буфер для сообщения об ошибке на случай, если будет найдена проблема

Если значение не было получено, вернуть ошибку

Если значение превышает максимально допустимую длину, вернуть ошибку

Оповещение вызывающей функции о том, что все прошло успешно

Функция IsCategoryIdInArray

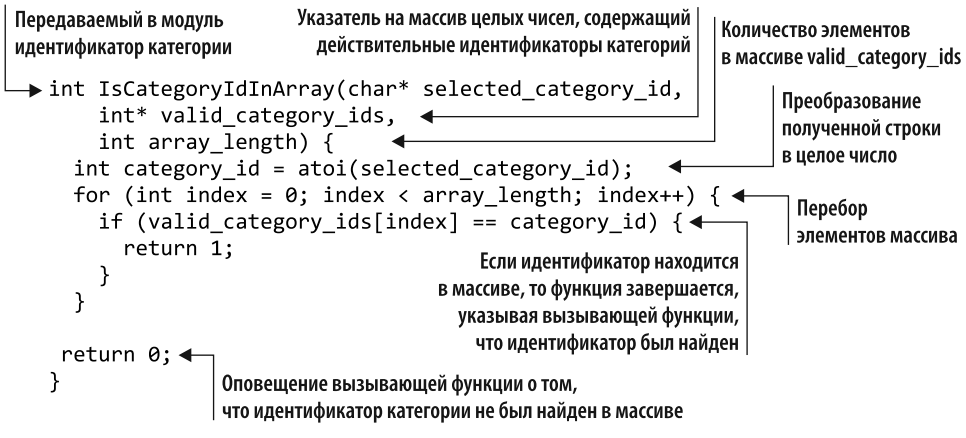
Прежде чем создавать функцию `ValidateCategory`, создадим вспомогательную функцию для упрощения логики. Эта вспомогательная функция получит название `IsCategoryIdInArray` и следующие параметры:

- идентификатор выбранной пользователем категории;

- указатель на массив целых чисел, содержащий допустимые идентификаторы категорий;
- количество элементов в массиве допустимых идентификаторов категорий.

Функция будет перебирать элементы в массиве, чтобы проверить, действительно ли выбранный пользователем идентификатор категории находится в массиве. Если да, то возвращается код `1` (успех). В противном случае возвращается код `0` (ошибка).

Добавьте следующую функцию `IsCategoryIdInArray` в файл `validate.cpp` после функции `ValidateName`:



Функция ValidateCategory

Последняя необходимая нам функция — это `ValidateCategory`, которая получит следующие параметры:

- идентификатор выбранной пользователем категории;
- указатель на массив целых чисел, содержащих допустимые идентификаторы категорий;
- количество элементов в массиве допустимых идентификаторов категорий;
- указатель на буфер, в который вы добавите сообщение об ошибке, если возникнет проблема с проверкой.

Функция проверит три условия:

- был ли предоставлен идентификатор категории? Вы убедитесь в этом, передав идентификатор вспомогательной функции `ValidateValueProvided`;
- был ли предоставлен указатель на массив допустимых идентификаторов категорий;

- находится ли выбранный пользователем идентификатор категории в массиве допустимых идентификаторов.

Если какая-либо из проверок не удалась, то помещаем соответствующее сообщение об ошибке в возвращаемый буфер и выходим из функции, вернув `0` (ошибка). Если код выполняется до конца функции, то это значит, что проблем с проверкой не было, поэтому возвращаем сообщение `1` (успех).

Добавьте функцию `ValidateCategory`, показанную в листинге 4.2, ниже функции `IsCategoryIdInArray` в файле `validate.cpp`.

Листинг 4.2. Функция ValidateCategory

```
...
#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
int ValidateCategory(char* category_id,
    int* valid_category_ids,
    int array_length,
    char* return_error_message) {
  if (ValidateValueProvided(category_id,
      "A Product Category must be selected.",
      return_error_message) == 0) {
    return 0;
  }

  if ((valid_category_ids == NULL) || (array_length == 0)) {
    strcpy(return_error_message,
        "There are no Product Categories available.");
    return 0;
  }

  if (IsCategoryIdInArray(category_id, valid_category_ids,
      array_length) == 0) {
    strcpy(return_error_message,
        "The selected Product Category is not valid.");
    return 0;
  }

  return 1;
}
```

Идентификатор выбранной категории, передаваемый в модуль

Указатель на массив целых чисел, содержащий действительные идентификаторы категорий

Количество элементов в массиве valid_category_ids

Буфер для сообщения об ошибке на случай, если будет найдена проблема

Если значение не было получено, вернуть ошибку

Если массив не был задан, вернуть ошибку

Если идентификатор заданной категории не был найден в массиве, вернуть ошибку

Оповещение вызывающей функции о том, что все прошло успешно

4.1.2. Компиляция кода в модуль WebAssembly

После того как код C++ будет изменен так, чтобы также мог быть скомпилирован с помощью Emscripten, можно сделать следующий шаг и дать указание Emscripten скомпилировать код в WebAssembly, как показано на рис. 4.5.

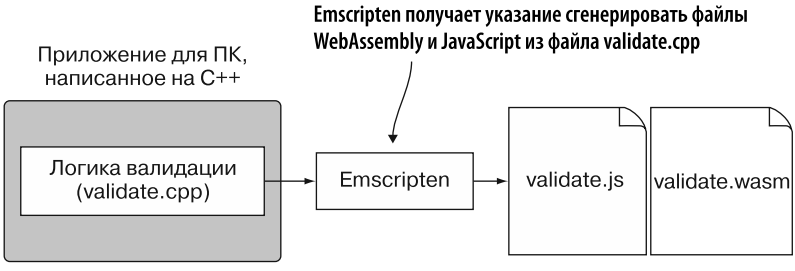


Рис. 4.5. Второй шаг процесса повторного использования кода на C++ — командуем Emscripten сгенерировать файлы WebAssembly и JavaScript

При написании JavaScript-кода для взаимодействия с модулем вы будете применять вспомогательные функции `ccall` и `UTF8ToString` в Emscripten (подробности о функции `ccall` см. в приложении Б). Чтобы добавить эти функции в сгенерированный файл JavaScript, вам необходимо указать их при компиляции кода на C++. Для этого воспользуйтесь массивом командной строки `EXTRA_EXPORTED_RUNTIME_METHODS`, указав эти функции.

ПРИМЕЧАНИЕ

При добавлении функций помните, что имена функций чувствительны к регистру. Например, функция UTF8ToString должна включать заглавные буквы UTF, T и S.

Чтобы скомпилировать код в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой сохранен файл `validate.cpp`, а затем выполните следующую команду:

```
emcc validate.cpp -o validate.js
➥ -s EXTRA_EXPORTED_RUNTIME_METHODS=['ccall','UTF8ToString']
```

4.1.3. Создание веб-страницы

После того как код на C++ будет изменен и скомпилирован в модуль WebAssembly, можно перейти к созданию страницы Edit Product (Изменить товар) для сайта, показанной на рис. 4.6.

СОВЕТ

Некоторые из вас хорошо владеют такими языками, как C или C++, но никогда по-настоящему не работали с HTML. Если вы хотите ознакомиться с основами HTML, то обратите внимание на несколько действительно хороших руководств, размещенных на следующем сайте: www.w3schools.com/html.

Чтобы веб-страница выглядела более профессионально, мы будем использовать Bootstrap вместо стилизации всего вручную. Этот популярный фреймворк для

веб-разработки включает ряд шаблонов дизайна, которые помогают упростить и ускорить разработку. В рамках целей данной книги вы будете просто указывать файлы, размещенные в CDN, но Bootstrap можно загрузить и включить в код вашей веб-страницы. Инструкции по загрузке Bootstrap находятся в приложении А.

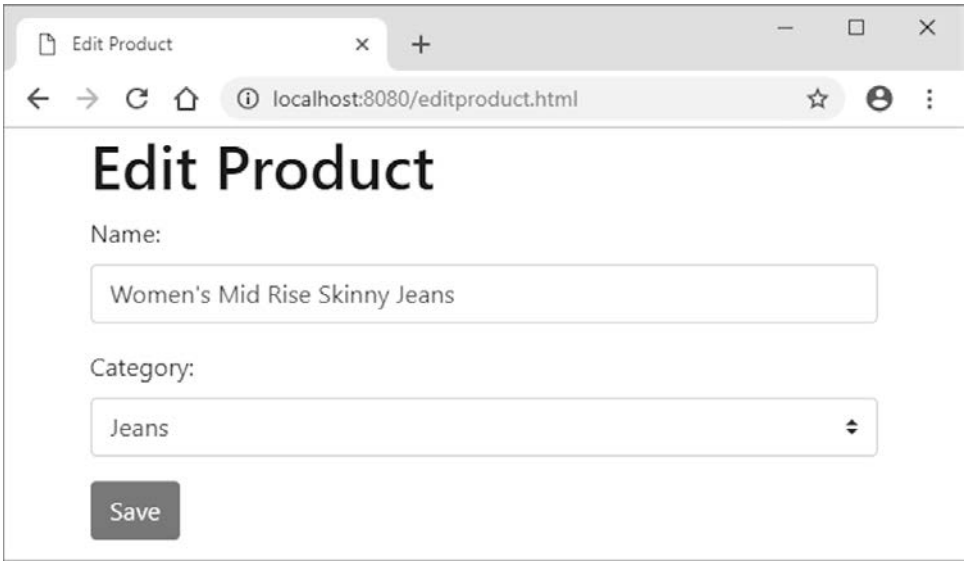


Рис. 4.6. Страница Edit Product (Изменить товар), созданием и валидацией которой мы займемся далее

СПРАВКА

Сеть доставки контента (content delivery network, CDN) географически распределена таким образом, чтобы обслуживать необходимые файлы, расположенные как можно ближе к устройству, которое их запрашивает. Это распределение позволяет получить файлы с сервера быстрее, благодаря чему сокращается время загрузки сайта.

В папке WebAssembly\Chapter 4\4.1 js_plumbing\ создайте папку frontend, а затем добавьте в нее файл `editproduct.html`. Откройте его в своем любимом текстовом редакторе и введите HTML-код, показанный в листинге 4.3.

Листинг 4.3. HTML-разметка страницы Edit Product (editproduct.html)

```
<!DOCTYPE html>
<html>
  <head>
    <title>Edit Product</title>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">
```

```
    <link rel="stylesheet"
    ➥ href="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.0/css/W3Schools
    ➥ bootstrap.min.css">
    <script
    ➥ src="https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/W3Schools
    ➥ jquery.min.js"></script>
    <script
    ➥ src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.0/umd/
    ➥ W3Schools popper.min.js"></script>
    <script
    ➥ src="https://maxcdn.bootstrapcdn.com/bootstrap/4.1.0/js/W3Schools
    ➥ bootstrap.min.js"></script>
  </head>
  <body onload="initializePage()">
    <div class="container">
      <h1>Edit Product</h1>

      <div id="errorMessage" class="alert alert-danger" role="alert"
      ➥ style="display:none;"></div>

      <div class="form-group">
        <label for="name">Name:</label>
        <input type="text" class="form-control" id="name">
      </div>

      <div class="form-group">
        <label for="category">Category:</label>
        <select class="custom-select" id="category">
          <option value="0"></option>
          <option value="100">Jeans</option>
          <option value="101">Dress Pants</option>
        </select>
      </div>

      <button type="button" class="btn btn-primary"
      ➥ onclick="onClickSave()">Save</button>
    </div>

    <script src="editproduct.js"></script>
    <script src="validate.js"></script>
  </body>
</html>
```

4.1.4. Написание кода на JavaScript, взаимодействующего с модулем

На рис. 4.7 показан следующий шаг процесса — копирование файлов, созданных Emscripten, `validate.js` и `validate.wasm`, в папку, в которой находится файл `editproduct.html`. Затем вы создадите файл `editproduct.js`, который

станет промежуточным звеном между пользователем, взаимодействующим с веб-страницей, и кодом, взаимодействующим с модулем.

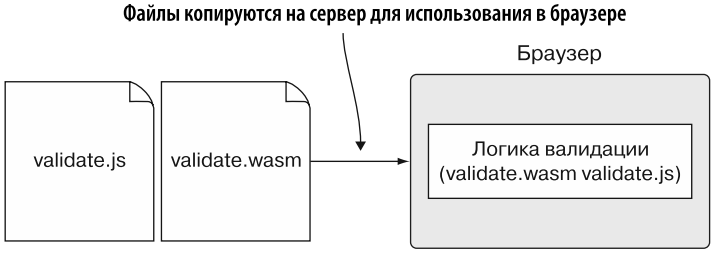


Рис. 4.7. Шаг 3 процесса повторного использования кода на C++ — копирование сгенерированных файлов в папку с HTML-файлом и создание JavaScript-кода для взаимодействия с модулем

Скопируйте файлы `validate.js` и `validate.wasm` из папки WebAssembly\Chapter 4\4.1 js_plumbing\source\ в папку WebAssembly\Chapter 4\4.1 js_plumbing\frontend\. В папке frontend создайте файл `editproduct.js` и откройте его.

Вместо того чтобы добавлять код для взаимодействия с сервером, мы будем имитировать получение данных с сервера, создав объект JavaScript `initialData`. Он послужит для инициализации элементов управления при отображении веб-страницы. Добавьте объект JavaScript в файл `editproduct.js`:

```
const initialData = {
  name: "Women's Mid Rise Skinny Jeans",
  categoryId: "100",
};
```

Имитирует данные, полученные с сервера

Функции `ValidateName` модуля во время вызова понадобится максимальное значение длины названия товара. Чтобы указать это значение, будем использовать константу `MAXIMUM_NAME_LENGTH`. Кроме того, пригодится массив допустимых идентификаторов категорий, `VALID_CATEGORY_IDS`, для применения при проверке идентификатора выбранной пользователем категории. Добавьте следующий фрагмент кода после объекта `initialData` в файл `editproduct.js`:

```
const MAXIMUM_NAME_LENGTH = 50;
const VALID_CATEGORY_IDS = [100, 101];
```

Максимально допустимая длина названия товара

Список допустимых идентификаторов категорий, которые можно выбрать

В HTML-коде страницы `editproduct.html` вы указали, что функция `initializePage` вызывается при загрузке веб-страницы. Этот вызов функции позволяет заполнять элементы управления на странице данными из объекта `initialData`.

В функции `initializePage` сначала заполняется поле названия товара значением имени в объекте `initialData`. Затем раскрывающийся список категорий перебирается, чтобы найти элемент в списке, который соответствует значению `categoryId` в объекте `initialData`. Если соответствующее значение `category ID` найдется, то нужный элемент в списке будет помечен как выбранный, а его индекс будет передан в свойство `selectedIndex`. Добавьте следующую функцию `initializePage` в файл `editproduct.js`:

```
function initializePage() {
  document.getElementById("name").value = initialData.name;

  const category = document.getElementById("category");
  const count = category.length;
  for (let index = 0; index < count; index++) {
    if (category[index].value === initialData.categoryId){
      category.selectedIndex = index;
      break;
    }
  }
}
```

Получает количество элементов в раскрывающемся списке

Перебирает все элементы в списке категорий

Если совпадение найдено, то выбрать этот элемент списка и выйти из цикла

Следующая функция, которую нужно добавить в файл `editproduct.js`, — это `getSelectedCategoryId`. Она возвращает идентификатор выбранного элемента из списка категорий и вызывается, когда пользователь нажимает кнопку Save (Сохранить):

```
function getSelectedCategoryId() {
  const category = document.getElementById("category");
  const index = category.selectedIndex;
  if (index !== -1) { return category[index].value; }

  return "0";
}
```

Если выбранный элемент находится в списке, вернуть значение элемента

Ничего не было выбрано, поэтому вернуть идентификатор, равный нулю

Теперь нужно создать функцию `setErrorMessage`, которая используется для показа пользователю сообщения об ошибке. Вы сделаете это, заполнив раздел веб-страницы строкой, полученной от модуля WebAssembly. Если в функцию передается пустая строка, то это значит, что на странице нужно скрыть раздел с ошибкой. В противном случае он отображается. Следующий фрагмент кода — это функция `setErrorMessage`, которую нужно добавить в файл `editproduct.js`:

```
function setErrorMessage(error) {
  const errorMessage = document.getElementById("errorMessage");
  errorMessage.innerText = error;
  errorMessage.style.display = (error === "" ? "none" : "");
}
```

В HTML-коде кнопки Save (Сохранить) на веб-странице указано событие `onclick` для запуска функции `onClickSave`, когда пользователь нажимает кнопку. В функции `onClickSave` введенные пользователем значения передаются функциям JavaScript `validateName` и `validateCategory`. Если какая-либо функция проверки указывает на наличие проблемы, то сообщение об ошибке модуля извлекается из памяти модуля и показывается пользователю.

СОВЕТ

Вы можете дать функциям JavaScript любое имя, но я назвал их таким образом, чтобы они соответствовали функции в вызываемом модуле. Например, функция JavaScript validateName вызывает функцию модуля ValidateName.

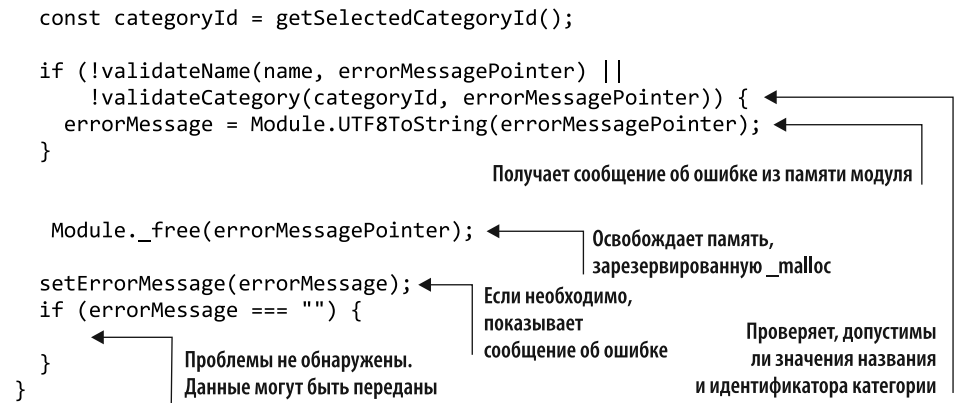
Как описано в предыдущих главах, модули WebAssembly поддерживают только четыре основных типа данных (32- и 64-битные целые числа, 32- и 64-битные числа с плавающей запятой). Для более сложных типов данных, таких как строки, необходимо использовать память модуля.

В Emscripten есть вспомогательная функция `ccall`, которая помогает вызывать функции модуля и управлять памятью при работе со строками, если предполагается, что эти строки будут использоваться только во время вызова. В таком случае в модуль передается буфер строк, чтобы его можно было заполнить соответствующей ошибкой проверки, если возникнет проблема с введенными пользователем данными. Поскольку память для хранения этой строки должна быть занята дольше, чем на время вызова функции модуля `ValidateName` или `ValidateCategory`, то нужно будет вручную управлять памятью в функции `onClickSave`. Для этого связующий код Emscripten предоставляет доступ к стандартным функциям `malloc` и `free` библиотеки C через функции `_malloc` и `_free` соответственно — с их помощью вы можете выделить и освободить память модуля.

Помимо выделения и освобождения памяти, также необходимо иметь возможность читать строку из памяти модуля. Для этого послужит вспомогательная функция Emscripten `UTF8ToString`. Она принимает указатель и считывает строку данного места в памяти. В листинге 4.4 приведена функция `onClickSave`, которую необходимо добавить в файл `editproduct.js` после функции `setErrorMessage`.

Листинг 4.4. Функция onClickSave в файле editproduct.js

```
Резервирует 256 байт памяти
модуля для сообщения об ошибке
...

                                    Получает введенные пользователем
function onClickSave() {                      значения с веб-страницы
  let errorMessage = "";
  const errorMessagePointer = Module._malloc(256);

  const name = document.getElementById("name").value;
```



Обращение к модулю: ValidateName

Первая вызываемая функция в модуле WebAssembly имеет следующую сигнатуру в C++:

```
int ValidateName(char* name,
    int maximum_length,
    char* return_error_message);
```

Для вызова функции `ValidateName` в модуле вы будете использовать вспомогательную функцию `ccall` в Emscripten. Для более подробной информации о параметрах функции `ccall` см. приложение Б. В функцию `ccall` будут переданы следующие значения в качестве параметров.

- `'ValidateName'` — определяет название вызываемой функции.
- `'number'` — возвращаемый тип, поскольку функция возвращает целое число.
- Массив со значениями `'string'`, `'number'` и `'number'` — типы данных параметров.

  Первый параметр функции `ValidateName` — символьный указатель `char*` для введенного названия товара. В этом случае допустимо недолговременное хранение строки, поэтому можно позволить функции `ccall` самостоятельно управлять памятью, указав значение `'string'` для данного параметра.

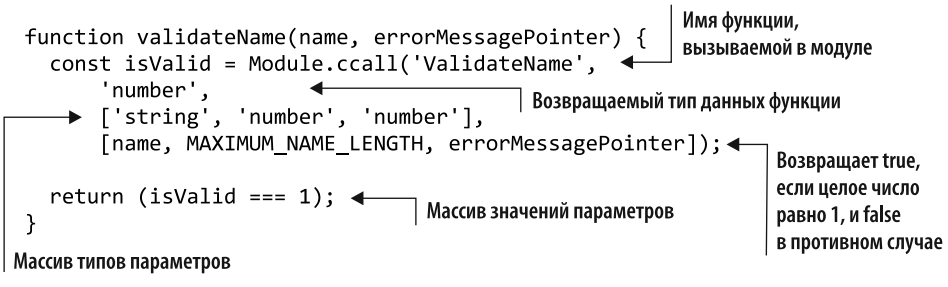
  Второй параметр ожидает `int`, поэтому достаточно указать тип `'number'`.

  Третий параметр может немного сбить с толку. Символьный указатель `char*` означает возвращаемое значение в случае ошибки. Этот указатель должен храниться долго, чтобы его можно было вернуть в JavaScript-функцию. В данном случае мы не можем позволить функции `ccall` управлять памятью, поэтому будем делать это самостоятельно в функции `onClickSave`. Строку

нужно передать как указатель, а для этого необходимо задать тип параметра как `'number'`.

- Массив, хранящий введенное пользователем значение названия товара, постоянное значение максимально допустимой длины названия товара и буфер для хранения возможного сообщения об ошибке.

В приведенном ниже фрагменте кода показана функция `validateName`, которую нужно добавить в файл `editproduct.js` после функции `onClickSave`:



СОВЕТ

В этом случае код для вызова функции ValidateName модуля довольно простой. Как вы увидите в следующих примерах, может использоваться более сложный код. Рекомендуется хранить код для каждой вызываемой функции WebAssembly в отдельной функции JavaScript, чтобы упростить сопровождение.

Обращение к модулю: ValidateCategory

Теперь напишем функцию `validateCategory` на JavaScript для вызова одноименной функции модуля. Она имеет следующую сигнатуру в C++:

```
int ValidateCategory(char* category_id,
    int* valid_category_ids,
    int array_length,
    char* return_error_message);
```

Функция `ValidateCategory` ожидает указатель на массив целых чисел, но тип параметра массива функции `ccall` предназначен только для 8-битных значений (дополнительную информацию об этих параметрах см. в приложении Б). Функция модуля ожидает массив из 32-битных целых чисел, поэтому необходимо вручную выделить память для массива и освободить ее, после того как будет получен результат вызова модуля.

Память модуля WebAssembly — это просто буфер типизированного массива. В Emscripten есть несколько представлений, которые позволяют просматривать память по-разному, чтобы было проще работать с разными типами данных.

Модуль ожидает массив целых чисел, поэтому будем использовать представление `HEAP32`.

Чтобы выделить достаточно памяти для указателя массива, вызов `Module._malloc` должен умножить количество элементов в массиве на количество байтов для каждого элемента, помещенного в объект `Module.HEAP32`. Для этого воспользуемся константой `Module.HEAP32.BYTES_PER_ELEMENT`, которая содержит значение `4` для объекта `HEAP32`.

После того как память для указателя массива выделена, вы можете использовать метод `set` объекта `HEAP32` в целях копирования содержимого массива в память модуля.

- Первый параметр — массив `VALID_CATEGORY_IDS`, который нужно скопировать в память модуля WebAssembly.
- Второй параметр — индекс, с которого метод `set` должен начать запись данных в базовый массив (память модуля). В этом случае, поскольку вы работаете с 32-битным представлением памяти, каждый индекс относится к одной из групп по 32 бита (4 байта). В результате нужно разделить адрес памяти на четыре.

Последняя функция JavaScript, которую необходимо добавить в конец файла `editproduct.js`, — функция `validateCategory`, представленная в листинге 4.5.

Листинг 4.5. Функция validateCategory в файле editproduct.js

```
...

function validateCategory(categoryId, errorMessagePointer) {
  const arrayLength = VALID_CATEGORY_IDS.length;
  const bytesPerElement = Module.HEAP32.BYTES_PER_ELEMENT;
  const arrayPointer = Module._malloc((arrayLength *
      bytesPerElement));
  Module.HEAP32.set(VALID_CATEGORY_IDS,
      (arrayPointer / bytesPerElement));

  const isValid = Module.ccall('ValidateCategory',
      'number',

      ['string', 'number', 'number', 'number'],
      [categoryId, arrayPointer, arrayLength, errorMessagePointer]);

  Module._free(arrayPointer);

  return (isValid === 1);
}
```

Получает количество байтов для каждого элемента объекта HEAP32

Выделяет достаточно памяти для каждого элемента массива

Копирует элементы массива в память модуля

Вызывает функцию ValidateCategory в модуле

Освобождает память, выделенную для массива

Возвращает true, если целое число равно 1, и false в противном случае

4.1.5. Просмотр результатов

Написав JavaScript-код, вы можете открыть браузер и ввести `http://localhost:8080/editproduct.html` в адресную строку, чтобы увидеть только что созданную веб-страницу. Вы можете протестировать валидацию, удалив весь текст из поля Name (Название) и нажав кнопку Save (Сохранить). На веб-странице должна появиться ошибка (рис. 4.8).

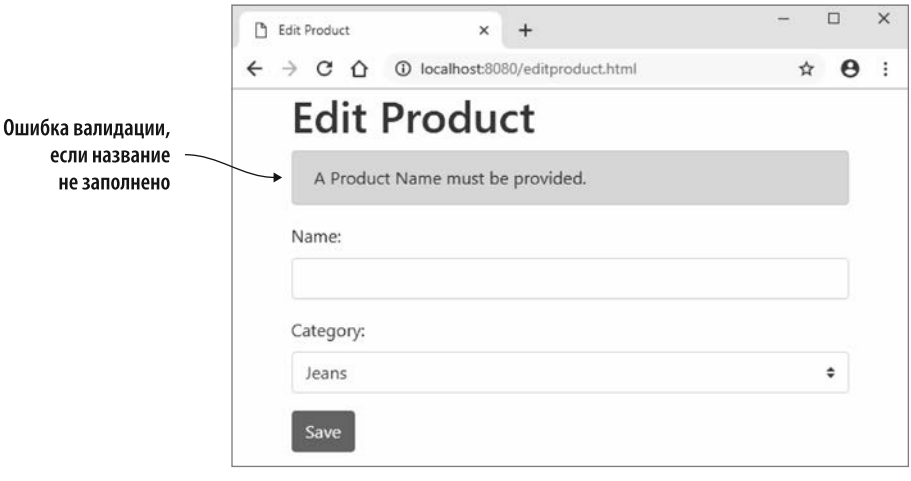


Рис. 4.8. Ошибка валидации при заполнении названия на странице Edit Product (Изменить товар)

4.2. ИСПОЛЬЗОВАНИЕ C ИЛИ C++ ДЛЯ СОЗДАНИЯ МОДУЛЯ БЕЗ EMSCRIPTEN

Предположим, вам нужно, чтобы Emscripten скомпилировал код C++ и не добавил какие-либо стандартные функции библиотеки C или не сгенерировал связующий файл JavaScript. Каким бы удобным ни был связующий код Emscripten, он скрывает многие детали работы с модулем WebAssembly. Этот подход полезен для обучения, поскольку вы получите возможность напрямую работать с такими вещами, как JavaScript WebAssembly API.

Обычно в производственном коде используется процесс, описанный в разделе 4.1, когда Emscripten включает стандартные функции библиотеки C, которые код использует в сгенерированном модуле. При этом Emscripten также создает связующий файл JavaScript, который обрабатывает загрузку и создание экземпляра модуля и включает вспомогательные функции, такие как `ccall`, чтобы упростить взаимодействие с модулем.

Как вы можете видеть на рис. 4.9, процесс в текущем разделе аналогичен процессу в разделе 4.1, за исключением того, что мы дадим Emscripten указание сгенерировать только файл WebAssembly, без связующего файла JavaScript.

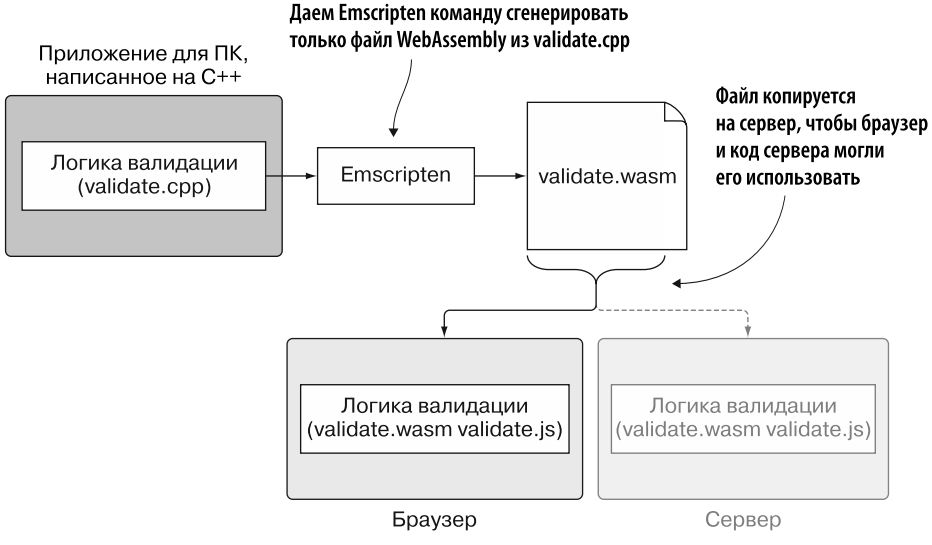


Рис. 4.9. Шаги по превращению существующей логики на C++ в WebAssembly для использования сайтом и кодом сервера, но без какого-либо сгенерированного Emscripten кода на JavaScript. Серверная часть, Node.js, будет рассмотрена в одной из следующих глав

4.2.1. Внесение изменений в C++

Хотя код в файле `validate.cpp`, созданном в разделе 4.1, довольно простой, он использует ряд стандартных функций библиотеки C — например, `strlen`, которые Emscripten не будет добавлять, если дать ему указание создать модуль в качестве вспомогательного модуля. Кроме того, поскольку код должен передавать указатели на значения, размещенные в памяти, нам нужен способ пометить эту память как заблокированную, чтобы код C или JavaScript не перезаписывал значения в данном разделе памяти, пока эта память не будет высвобождена в коде.

Рис. 4.10. Первым делом необходимо создать собственные версии нужных функций стандартной библиотеки C, чтобы код можно было скомпилировать с помощью Emscripten

У вас не будет доступа к функциям `malloc` и `free` стандартной библиотеки, поэтому первым делом (рис. 4.10) нужно создать собственные функции для этого.

Заголовочный файл для системных функций вспомогательного модуля

Создайте папку WebAssembly\Chapter 4\4.2 side_module\source\. В папке source создайте файл `side_module_system_functions.h` и откройте его в вашем редакторе. Добавьте следующий фрагмент кода в файл, чтобы определить сигнатуры функций, которые мы создадим далее:

```
#pragma once

#ifndef SIDE_MODULE_SYSTEM_FUNCTIONS_H_
#define SIDE_MODULE_SYSTEM_FUNCTIONS_H_

#include <stdio.h>

void InsertIntoAllocatedArray(int new_item_index, int offset_start,
    int size_needed);

int create_buffer(int size_needed);
void free_buffer(int offset);

char* strcpy(char* destination, const char* source);
size_t strlen(const char* value);

int atoi(const char* value);

#endif // SIDE_MODULE_SYSTEM_FUNCTIONS_H_
```

Файл реализации для системных функций вспомогательного модуля

Теперь создайте файл `side_module_system_functions.cpp` в исходной папке и откройте его в редакторе. Создадим простую замену функций `malloc` и `free` стандартной библиотеки C. Функция `malloc` находит первую доступную ячейку памяти, достаточно большую для запрошенного размера памяти. Затем она помечает этот блок памяти, чтобы он не использовался другими запросами кода для памяти. Как только код завершает работу с блоком памяти, вызывается функция `free` стандартной библиотеки C, чтобы снять блокировку.

Будем использовать массив для обработки выделения фрагментов памяти для десяти одновременных запросов, чего более чем достаточно для этого кода валидации. У вас всегда должна быть хотя бы одна страница памяти размером 65 536 байт (64 Кбайт), и выделение памяти будет происходить внутри данного блока.

В начале файла `side_module_system_functions.cpp` добавьте строки `#include` для стандартной библиотеки ввода и вывода C и заголовочного файла Emscripten.

Добавьте открывающий блок `extern "C"`, а затем добавьте константы для размера памяти и максимального допустимого количества параллельных блоков памяти:

```
#include <stdio.h>
#include <emscripten.h>

#ifdef __cplusplus
extern "C" {
#endif

const int TOTAL_MEMORY = 65536;
const int MAXIMUM_ALLOCATED_CHUNKS = 10;
```

После констант добавьте переменную `current_allocated_count`, которая будет указывать, сколько блоков памяти выделено в данный момент. Добавьте определение для объекта `MemoryAllocated`, в котором будут храниться начало выделенной памяти и длина блока памяти. Затем создайте массив, который будет содержать объекты, указывающие, какие блоки памяти используются:

```
int current_allocated_count = 0;

struct MemoryAllocated {
  int offset;
  int length;
};

struct MemoryAllocated
➥ AllocatedMemoryChunks[MAXIMUM_ALLOCATED_CHUNKS];
```

Следующее, что нужно сделать, — создать функцию, которая будет принимать индекс, в который будет добавлен новый блок памяти в массиве `Allocated-MemoryChunks`. Любые элементы в массиве от этого индекса до конца массива будут перемещены на одну позицию в сторону конца массива. Затем функция поместит начальную позицию (смещение) блока памяти и размер блока памяти в запрошенное место в массиве. Добавьте код, показанный в листинге 4.6, после массива `AllocatedMemoryChunks` в файле `side_module_system_functions.cpp`.

Листинг 4.6. Функция InsertIntoAllocatedArray

```
...

void InsertIntoAllocatedArray(int new_item_index, int offset_start,
    int size_needed) {
  for (int i = (MAXIMUM_ALLOCATED_CHUNKS – 1); i > new_item_index; i--){
    AllocatedMemoryChunks[i] = AllocatedMemoryChunks[(i - 1)];
  }

  AllocatedMemoryChunks[new_item_index].offset = offset_start;
  AllocatedMemoryChunks[new_item_index].length = size_needed;

  current_allocated_count++;
}
```

Теперь создайте упрощенную версию функции `malloc` под названием `create_buffer`. При добавлении строковых литералов в код на C++ и компиляции кода в модуль WebAssembly Emscripten автоматически загружает эти строковые литералы в память модуля при создании экземпляра модуля. Из-за этого код должен будет оставить место для строк и начнет выделять память только с 1024-го байта. Код также увеличит размер запрошенной памяти так, чтобы он был кратным восьми.

Первое, что сделает код, — просмотрит текущую выделенную память, чтобы увидеть, есть ли свободное место между выделенными блоками, соответствующее запрошенному размеру памяти. Если да, то новый выделенный блок будет вставлен в массив по этому индексу. Если между существующими выделенными блоками памяти недостаточно места для запрошенного размера памяти, то код проверяет наличие места после выделенной в данный момент памяти.

Код вернет смещение памяти того места, в котором был выделен блок памяти, если таковое имеется. В противном случае вернет `0` (ноль), что будет указывать на ошибку, учитывая, что код начнет выделять память только с 1024-го байта.

Добавьте код, показанный в листинге 4.7, в файл `side_module_system_functions.cpp`.

Листинг 4.7. Упрощенная версия функции malloc

```
...

EMSCRIPTEN_KEEPALIVE
int create_buffer(int size_needed) {
 if (current_allocated_count == MAXIMUM_ALLOCATED_CHUNKS) { return 0; }

 int offset_start = 1024;
 int current_offset = 0;
 int found_room = 0;

 int memory_size = size_needed;
 while (memory_size % 8 != 0) { memory_size++; }

 for (int index = 0; index < current_allocated_count; index++) {
   current_offset = AllocatedMemoryChunks[index].offset;
   if ((current_offset - offset_start) >= memory_size) {
     InsertIntoAllocatedArray(index, offset_start, memory_size);
     found_room = 1;
     break;
   }

   offset_start = (current_offset + AllocatedMemoryChunks[index].length);
 }

 if (found_room == 0) {
   if (((TOTAL_MEMORY - 1) - offset_start) >= size_needed) {
     AllocatedMemoryChunks[current_allocated_count].offset = offset_start;
     AllocatedMemoryChunks[current_allocated_count].length = size_needed;
```

Увеличивает размер так, чтобы следующее смещение было кратно восьми

Есть ли свободное место между текущими выделенными блоками памяти?

Есть ли свободное место между последним блоком памяти и концом памяти модуля?

Место не было найдено между текущими выделенными блоками памяти

```
    current_allocated_count++;
    found_room = 1;
  }
 }

 if (found_room == 1) { return offset_start; }
 return 0;
}
```

Ваш эквивалент функции `free` будет называться `free_buffer`. В этой функции вы просто просматриваете массив выделенных блоков памяти, пока не найдете смещение, полученное от вызывающей стороны. Найдя данный элемент массива, вы переместите все элементы после него на одну позицию в сторону начала массива. Добавьте код, показанный в листинге 4.8, после функции `create_buffer`.

Листинг 4.8. Упрощенная версия функции free

```
...

EMSCRIPTEN_KEEPALIVE
void free_buffer(int offset) {
  int shift_item_left = 0;

  for (int index = 0; index < current_allocated_count; index++) {
    if (AllocatedMemoryChunks[index].offset == offset) {
      shift_item_left = 1;
    }

    if (shift_item_left == 1) {
      if (index < (current_allocated_count - 1)) {
        AllocatedMemoryChunks[index] = AllocatedMemoryChunks[(index + 1)];
      }
      else {
        AllocatedMemoryChunks[index].offset = 0;
        AllocatedMemoryChunks[index].length = 0;
      }
    }
  }

  current_allocated_count--;
}
```

В следующем фрагменте кода в файле `side_module_system_functions.cpp` создается версия функций `strcpy` и `strlen` системной библиотеки:

```
char* strcpy(char* destination, const char* source) {
  char* return_copy = destination;
  while (*source) { *destination++ = *source++; }
  *destination = 0;

  return return_copy;
```

```
}

size_t strlen(const char* value) {
  size_t length = 0;
  while (value[length] != '\0') { length++; }

  return length;
}
```

В листинге 4.9 в файле `side_module_system_functions.cpp` создается версия функции `atoi` системной библиотеки.

Листинг 4.9. Версия функции atoi

```
...

int atoi(const char* value) {
  if ((value == NULL) || (value[0] == '\0')) { return 0; }

  int result = 0;
  int sign = 0;

  if (*value == '-') { sign = -1; ++value; }

  char current_value = *value;
  while (current_value != '\0') {
    if ((current_value >= '0') && (current_value <= '9')) {
      result = result * 10 + current_value - '0';
      ++value;
      current_value = *value;
    }
    else {
      return 0;
    }
  }

  if (sign == -1) { result *= -1; }
  return result;
}
```

Сделать пометку, если первый символ — это минус. Перейти к следующему байту

Перебирать массив, пока не будет достигнут нулевой ограничитель

Если текущий символ — цифра…

Переместить указатель на следующий байт

…конвертировать current_value в целое число. Добавить его к result

Если был найден нецифровой знак — завершить работу, вернуть ноль

Если было найдено отрицательное число, то изменить значение на положительное

Наконец, добавьте закрывающую фигурную скобку для блока `extern "C"` в конце файла `side_module_system_functions.cpp`, как показано в следующем фрагменте кода:

```
#ifdef __cplusplus
}
#endif
```

Теперь, завершив работу с файлом `side_module_system_functions.cpp`, скопируйте файл `validate.cpp` из папки WebAssembly\Chapter 4\4.1 js_plumbing\source\ и поместите его в папку WebAssembly\Chapter 4\4.2 side_module\source\.

Откройте файл `validate.cpp` и удалите строки `#include` для файлов `cstdlib` и `cstring`. Затем добавьте `#include` для нового заголовочного файла `side_module_system_functions.h` перед функцией `ValidateValueProvided` и внутри блока `extern "C"`.

ПРЕДУПРЕЖДЕНИЕ

Добавляемый заголовочный файл должен быть помещен в блок extern "C". Это связано с тем, что компилятору Emscripten нужно будет скомпилировать два файла .cpp. Хотя функции обоих файлов находятся внутри блоков extern "C", компилятор Emscripten по-прежнему предполагает, что вызовы функций в файле validate.cpp компилируются в файл C++, где имена функций были изменены. Компилятор не увидит измененные имена функций в сгенерированном модуле и будет считать, что вместо этого их нужно импортировать.

В следующем фрагменте кода показаны изменения в файле `validate.cpp`:

```
#ifdef __EMSCRIPTEN__
  #include <emscripten.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif

#include "side_module_system_functions.h"
```

Важно: поместите заголовочный файл в блок extern "C"

4.2.2. Компиляция кода в модуль WebAssembly

После того как код на C++ будет создан, можно сделать следующий шаг — дать Emscripten команду скомпилировать код в модуль WebAssembly, но без встроенного JavaScript-кода, как показано на рис. 4.11.

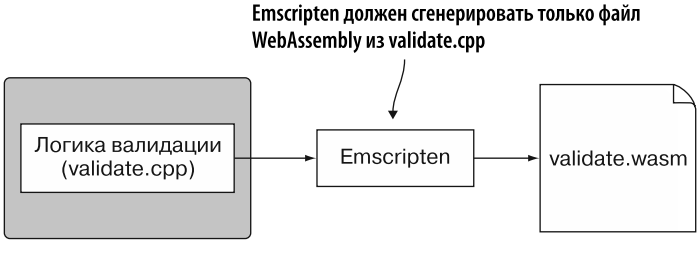


Рис. 4.11. Шаг 2 процесса — команда Emscripten сгенерировать только файл WebAssembly. В этом случае Emscripten не будет генерировать связующий файл JavaScript

Чтобы скомпилировать код на C++ в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой вы сохранили файлы C++, и выполните следующую команду:

```
emcc side_module_system_functions.cpp validate.cpp -s SIDE_MODULE=2
➥ -O1 -o validate.wasm
```

4.2.3. Создание файла JavaScript, взаимодействующего с модулем

После того как модуль WebAssembly будет получен, можно сделать следующий шаг (рис. 4.12). В папке WebAssembly\Chapter 4\4.2 side_module\ создайте папку frontend и скопируйте в нее файлы `editproduct.html` и `editproduct.js` из папки WebAssembly\Chapter 4\4.1 js_plumbing\frontend\.

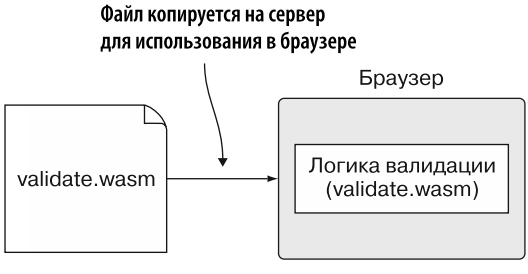


Рис. 4.12. Шаг 3 процесса — копирование сгененерированного файла в папку с HTML-файлом и написание кода на JavaScript для взаимодействия с модулем

Затем скопируйте файл `validate.wasm` из папки WebAssembly\Chapter 4\4.2 side_module\source\ в новую папку frontend. Первое, что нужно сделать, — открыть файл `editproduct.html` и удалить ссылку на JavaScript-файл `validate.js` в самом низу. Конец файла `editproduct.html` теперь должен выглядеть следующим образом:

```
    </div>

    <script src="editproduct.js"></script>
  </body>
</html>
```

Затем внесите несколько изменений в файл `editproduct.js` (листинг 4.10): добавьте две глобальные переменные, `moduleMemory` и `moduleExports`, перед функцией `initializePage`. Переменная `moduleMemory` хранит ссылку на объект `WebAssembly.Memory` модуля, чтобы можно было читать и записывать в память.

Поскольку у вас нет доступа к связующему коду Emscripten, у вас нет и объекта `Module`. Вместо этого используйте глобальную ссылку на объект `moduleExports`, которую вы получите при создании экземпляра модуля. Ссылка на `moduleExports`

позволит вызывать все экспортируемые функции в модуле. Добавьте также код в конце функции `initializePage` для загрузки и создания экземпляра модуля.

Листинг 4.10. Изменения в функции initializePage в файле editproduct.js

```
...

let moduleMemory = null;
let moduleExports = null;

function initializePage() {
  ...

  moduleMemory = new WebAssembly.Memory({initial: 256});

  const importObject = {
    env: {
      __memory_base: 0,
      memory: moduleMemory,
    }
  };

  WebAssembly.instantiateStreaming(fetch("validate.wasm"),
  ➥ importObject).then(result => {
    moduleExports = result.instance.exports;
  });
}
...
```

Добавляются две новые глобальные переменные

Ссылка на память модуля помещается в глобальную переменную

Загружает и создает экземпляр модуля

Хранит ссылку на экспортируемые функции экземпляра модуля

Компилятор Emscripten помещает символ подчеркивания перед каждой функцией в модуле, поэтому функции модуля, такие как `create_buffer`, будут иметь символ подчеркивания в начале имени в листинге 4.11.

Следующая функция, которую нужно изменить, — `onClickSave`, где вызов `Module._malloc` будет заменен на `moduleExports._create_buffer`, вызов `Module.UTF8ToString` — на `getStringFromMemory`, а вызов `Module._free` — на `moduleExports._free_buffer`. Изменения функции `onClickSave` выделены жирным шрифтом в листинге 4.11.

Память, переданная модулю WebAssembly во время инициализации, была предоставлена через объект `WebAssembly.Memory`, на который вы сохранили ссылку в переменной `moduleMemory`. Внутри объекта `WebAssembly.Memory` содержится объект `ArrayBuffer`, который предоставляет байты для модуля в целях имитации реальной машинной памяти. Вы можете получить доступ к базовому объекту `ArrayBuffer`, содержащемуся в ссылке `moduleMemory`, обращаясь к свойству `buffer`.

Как вы помните, в программном коде Emscripten есть такие объекты, как `HEAP32`, которые позволяют просматривать память модуля (`ArrayBuffer`) по-разному, чтобы было проще работать с разными типами данных. Без доступа к внутреннему коду

Emscripten у вас нет доступа к таким объектам, как `HEAP32`, но, к счастью, эти объекты просто ссылаются на объекты JavaScript, такие как `Int32Array`, к которым доступ есть.

Листинг 4.11. Изменения функции onClickSave в файле editproduct.js

```
...

function onClickSave() {
  let errorMessage = "";
  const errorMessagePointer = moduleExports._create_buffer(256);

  const name = document.getElementById("name").value;
  const categoryId = getSelectedCategoryId();

  if (!validateName(name, errorMessagePointer) ||
      !validateCategory(categoryId, errorMessagePointer)) {
    errorMessage = getStringFromMemory(errorMessagePointer);
  }

  moduleExports._free_buffer(errorMessagePointer);

  setErrorMessage(errorMessage);
  if (errorMessage === "") {

  }
}
...
```

Заменяет Module._malloc на moduleExports._create_buffer

Заменяет Module.UTF8ToString на вспомогательную функцию для чтения строки из памяти

Заменяет Module._free на moduleExports._free_buffer

Проблем при валидации не было обнаружено, данные могут быть сохранены

Необходимо создать вспомогательную функцию `getStringFromMemory`, которая будет считывать строки, возвращаемые коду JavaScript из памяти модуля. Строки в C или C++ помещаются в память как массив 8-битных символов, поэтому мы будем использовать объект JavaScript `Uint8Array` для доступа к памяти модуля, начиная со смещения, определенного указателем. Получив представление, перебирайте элементы в массиве, считывая по одному символу за раз, пока не дойдете до символа нулевого ограничителя.

Добавьте вспомогательную функцию `getStringFromMemory`, показанную в листинге 4.12, после функции `onClickSave` в файле `editproduct.js`.

Теперь, имея возможность читать строку из памяти модуля, вам нужно создать функцию, которая позволит записывать строку в память модуля. Подобно функции `getStringFromMemory`, функция `copyStringToMemory` начинается с создания объекта `Uint8Array` для управления памятью модуля. Затем мы воспользуемся объектом `TextEncoder` в JavaScript, чтобы превратить строку в массив байтов. Получив его из строки, можно вызвать метод `set` объекта `Uint8Array`, передав массив байтов для первого параметра и смещение, с которого следует начать запись этих байтов в качестве второго параметра.

Листинг 4.12. Функция getStringFromMemory в файле editproduct.js

```
...

function getStringFromMemory(memoryOffset) {
  let returnValue = "";

  const size = 256;
  const bytes = new Uint8Array(moduleMemory.buffer, memoryOffset, size);

  let character = "";
  for (let i = 0; i < size; i++) {
    character = String.fromCharCode(bytes[i]);
    if (character === "\0") { break; }

    returnValue += character;
  }

  return returnValue;
}
```

Получает раздел памяти, начиная со смещения и заканчивая следующими 256 символами

Перебирает байты по одному

Конвертирует текущий байт в символ

Если текущий символ — нулевой ограничитель, значит, чтение строки завершено

Добавляет текущий символ в возвращаемую строку до перехода к следующему символу

Ниже приводится функция `copyStringToMemory`, которую необходимо добавить в файл `editproduct.js` после функции `getStringFromMemory`:

```
function copyStringToMemory(value, memoryOffset) {
  const bytes = new Uint8Array(moduleMemory.buffer);
  bytes.set(new TextEncoder().encode((value + "\0")),
      memoryOffset);
}
```

Измените функцию `validateName` так, чтобы она сначала выделяла память для названия товара, введенного пользователем. Скопируйте строковое значение в память модуля, в ячейку памяти указателя, вызвав функцию `copyStringToMemory`. Затем вызовите функцию модуля `_ValidateName`; после этого освободите память, выделенную для указателя на имя.

В следующем фрагменте кода показана измененная функция `validateName`:

```
function validateName(name, errorMessagePointer) {
  const namePointer = moduleExports._create_buffer(
      (name.length + 1));
  copyStringToMemory(name, namePointer);

  const isValid = moduleExports._ValidateName(namePointer,
      MAXIMUM_NAME_LENGTH, errorMessagePointer);

  moduleExports._free_buffer(namePointer);

  return (isValid === 1);
}
```

Последний элемент, который нужно изменить, — это функция `validateCategory`. Мы выделим память для идентификатора категории, а затем скопируем идентификатор в ячейку памяти по указателю.

Функция выделит память, необходимую для элементов, в глобальном массиве `VALID_CATEGORY_IDS`, а затем скопирует каждый элемент массива в память модуля аналогично подходу, который использовался в коде подключения Emscripten. Разница в том, что у нас нет доступа к объекту `HEAP32` в Emscripten, но этот объект является просто ссылкой на объект `Int32Array` в JavaScript, к которому можно получить доступ.

После того как значения массива будут скопированы в память модуля, код вызывает функцию модуля `_ValidateCategory`. Когда она возвращает результат, код освобождает память, выделенную для указателей на массив и строку. В листинге 4.13 показана измененная функция `validateCategory`.

Листинг 4.13. Измененная функция validateCategory

```
 ...

 function validateCategory(categoryId, errorMessagePointer) {
   const categoryIdPointer = moduleExports._create_buffer(
 ➥ (categoryId.length + 1));
   copyStringToMemory(categoryId, categoryIdPointer);

   const arrayLength = VALID_CATEGORY_IDS.length;
   const bytesPerElement = Int32Array.BYTES_PER_ELEMENT;
   const arrayPointer = moduleExports._create_buffer(
   ➥ (arrayLength * bytesPerElement));

   const bytesForArray = new Int32Array(moduleMemory.buffer);
   bytesForArray.set(VALID_CATEGORY_IDS, (arrayPointer / bytesPerElement));

   const isValid = moduleExports._ValidateCategory(categoryIdPointer,
   ➥ arrayPointer, arrayLength, errorMessagePointer);

   moduleExports._free_buffer(arrayPointer);
   moduleExports._free_buffer(categoryIdPointer);

   return (isValid === 1);
 }
```

Выделяет память для идентификатора категории

Копирует идентификатор в память модуля

Выделяет память для каждого элемента массива

Получает память в виде Int32Array и копирует значения в массив

Вызывает функцию _ValidateCategory модуля

Освобождает выделенную память

4.2.4. Просмотр результатов

Внеся все изменения в код, вы можете открыть браузер и ввести в адресную строку `http://localhost:8080/editproduct.html`, чтобы просмотреть веб-страницу. Вы

можете проверить валидацию, добавив более 50 символов в поле Name (Название) и нажав кнопку Save (Сохранить), которая должна отобразить ошибку валидации, как показано на рис. 4.13.

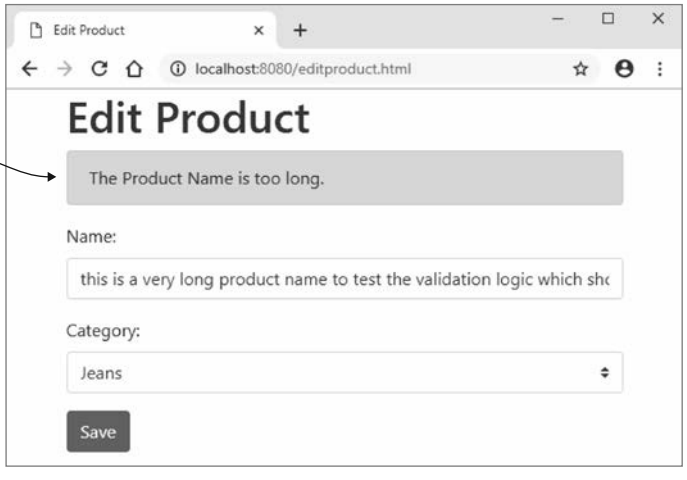


Рис. 4.13. Ошибка валидации названия на странице Edit Product (Изменить товар) при слишком длинном названии

А теперь — как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже приведены некоторые возможные варианты использования того, что вы узнали в этой главе.

- Вы можете адаптировать вашу программу на C++ или взять часть кода и скомпилировать ее в WebAssembly, чтобы ее можно было запустить в браузере.
- Если у вас есть код на JavaScript, который вызывает сервер или сторонний API и получает в ответ большие объемы текстовых данных, вы можете создать модуль WebAssembly, который будет парсить строку для получения данных, необходимых вашей веб-странице.
- Если у вас есть сайт, который позволяет пользователям загружать фотографии, вы можете создать модуль WebAssembly, принимающий байты файла для изменения размера или сжатия фотографии перед загрузкой. Благодаря этому можно улучшить пропускную способность, что помогло бы пользователю сократить использование данных и уменьшило бы время обработки на сервере.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какие два способа позволяют Emscripten сделать ваши функции видимыми для JavaScript-кода?
2. Как предотвратить искажение имен функций при компиляции, чтобы код JavaScript мог использовать ожидаемое имя функции?

РЕЗЮМЕ

В этой главе вы углубились в аспект повторного использования кода WebAssembly, создав веб-страницу, принимающую пользовательскую информацию, которая должна быть проверена.

- Используя символ условной компиляции `__EMSCRIPTEN__` и поместив функции в блок `extern "C"`, вы можете адаптировать существующий код так, чтобы его также можно было скомпилировать через Emscripten. Это позволяет использовать один и тот же код на C или C++, который может быть, например, частью приложения для ПК. Он также может быть доступным для использования в браузере или в Node.js.
- Включив объявление `EMSCRIPTEN_KEEPALIVE` в функции, вы можете автоматически добавить функцию в список функций Emscripten, видимых JavaScript-коду. При использовании этого объявления вам не нужно добавлять функцию в массив `EXPORTED_FUNCTIONS` в командной строке при компиляции модуля.
- Вы можете вызывать функции модуля с помощью вспомогательной функции `ccall` в Emscripten.
- Передача чего-либо, помимо целого числа или числа с плавающей запятой между модулем и кодом JavaScript, требует взаимодействия с памятью модуля. JavaScript-код, созданный Emscripten, предоставляет для этого ряд полезных функций.

5 Создание модуля WebAssembly, вызывающего JavaScript

В этой главе

- ✓ Вызов JavaScript напрямую с помощью инструментов Emscripten.
- ✓ Вызов JavaScript без использования инструментов Emscripten.

В главе 4 вы создали модуль WebAssembly, который вызывал JavaScript-код с помощью вспомогательной функции `ccall` из Emscripten. Буфер передавался в качестве параметра функции модуля, чтобы в случае возникновения проблемы можно было вернуть сообщение об ошибке, поместив его в буфер. Если возникает проблема, то JavaScript считывает строку из памяти модуля и затем отображает сообщение пользователю, как показано на рис. 5.1.

Представьте, что вместо передачи буфера функции модуля в случае возникновения проблемы модуль может просто передать сообщение об ошибке непосредственно в ваш JavaScript-код, как показано на рис. 5.2.

Используя набор инструментов Emscripten, вы можете взаимодействовать с кодом JavaScript из вашего модуля тремя способами.

1. Использовать макросы `Emscripten`. К ним относятся серия макросов `emscripten_run_script`, макрос `EM_JS` и серия макросов `EM_ASM`.
2. Добавить собственный JavaScript-код в файл JavaScript в Emscripten, который можно использовать напрямую.

3. Задействовать указатели на функции, в которых код JavaScript указывает функцию, вызываемую модулем. Мы рассмотрим этот подход в главе 6.

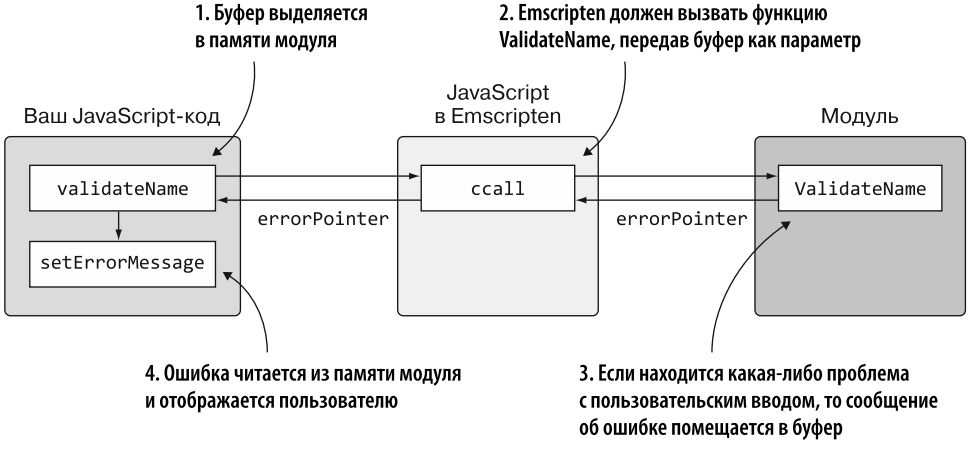


Рис. 5.1. Взаимодействие кода на JavaScript с функциями модуля

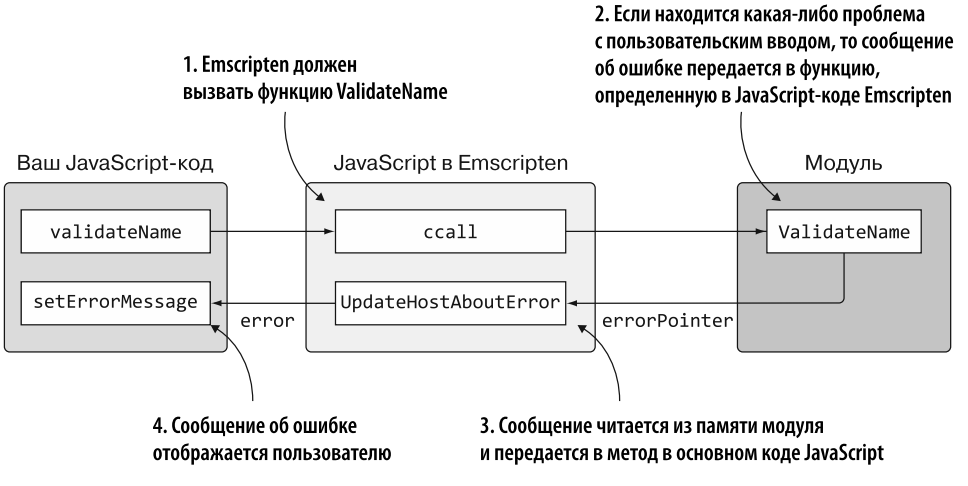


Рис. 5.2. Вызов модулем функции из JavaScript-кода

При любом способе взаимодействия с JavaScript из модуля один подход может работать лучше, чем другой, в определенных обстоятельствах.

1. Макросы Emscripten могут быть весьма полезны при отладке или когда нужно только взаимодействие с кодом JavaScript напрямую. По мере увеличения сложности кода макроса или количества взаимодействий с JavaScript вы мо-

жете рассмотреть возможность отделения кода макроса от кода на C или C++. Это следует сделать, чтобы упростить сопровождение кода и вашего модуля, и веб-страницы.

Когда используются серии макросов `EM_JS` и `EM_ASM`, на самом деле компилятор Emscripten создает необходимые функции и добавляет их в сгенерированный файл JavaScript в Emscripten. Вызывая макросы, модуль WebAssembly в действительности вызывает сгенерированные функции JavaScript.

СПРАВКА

Более подробную информацию о макросах Emscripten, в том числе о том, как их использовать, можно найти в приложении B.

2. Как вы увидите в данной главе, вызывать JavaScript напрямую несложно, и это несколько упростит JavaScript вашего сайта. Если вы планируете вызывать функции из функции JavaScript, размещенной в сгенерированном Emscripten JavaScript-коде, то вам необходимо иметь какое-то представление об основном коде JavaScript. Если вы поставляете модуль третьей стороне, то потребуются четкие инструкции по правильной настройке модуля, чтобы не было ошибок, — например, о том, что функция не существует.

ПРЕДУПРЕЖДЕНИЕ

Если вы планируете использовать этот подход вместе с Node.js, то JavaScript-код, который вы добавляете в сгенерированный файл JavaScript, должен быть автономным. Работа с Node.js рассматривается более подробно в главе 10, но, по сути, из-за того, как Node.js загружает файл JavaScript в Emscripten, код в этом файле не может вызывать ваш основной JavaScript-код.

3. В главе 6 вы увидите, что использование указателей на функции дает гораздо больше гибкости, поскольку модулю не нужно знать, какие функции существуют в вашем JavaScript-коде. Вместо этого модуль просто вызовет предоставленную JavaScript-функцию. Дополнительная гибкость указателей на функции связана с немного большей сложностью, поскольку такой подход требует большего количества кода на JavaScript, чтобы все работало правильно.

Вместо того чтобы позволять Emscripten создавать функции JavaScript с помощью макросов, вы можете определить собственный JavaScript, который будет включен в файл JavaScript в Emscripten. Мы изучим такой подход в данной главе.

В рамках этого сценария мы изменим модуль валидации, созданный в главе 4, чтобы при возникновении проблемы при валидации сообщение об ошибке не передавалось обратно вызывающей функции с помощью параметра. Вместо этого вы сделаете следующее (рис. 5.3).

1. Если была найдена проблема с пользовательским вводом, то пусть модуль вызовет функцию JavaScript, которую вы поместите в созданный Emscripten файл JavaScript.
2. Функция JavaScript будет принимать указатель от модуля и считывать сообщение об ошибке из памяти модуля.
3. Затем она передаст сообщение в основной JavaScript веб-страницы, где будет обработано обновление пользовательского интерфейса с полученной ошибкой.

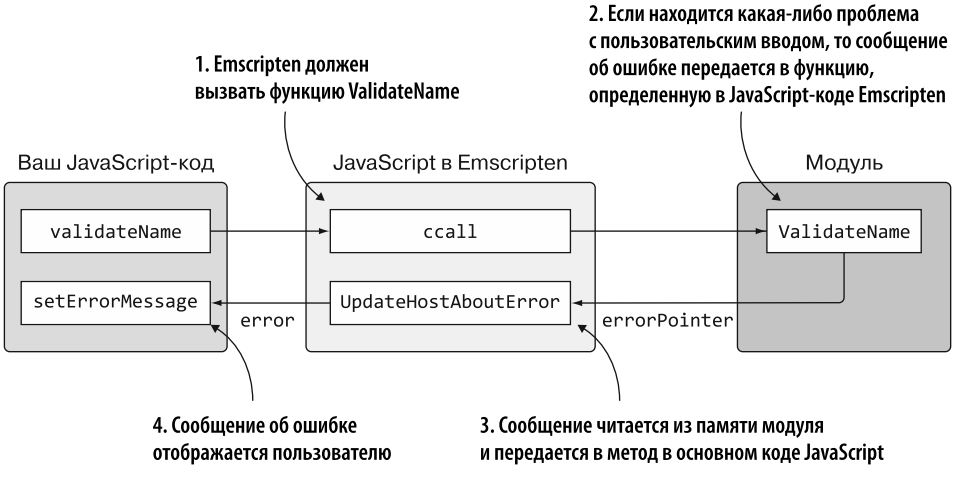


Рис. 5.3. Изменения в модуле и JavaScript, которые позволят модулю вызывать JavaScript напрямую

5.1. ИСПОЛЬЗОВАНИЕ C ИЛИ C++ ДЛЯ СОЗДАНИЯ МОДУЛЯ СО СВЯЗУЮЩИМ КОДОМ EMSCRIPTEN

Вернемся к логике валидации в C++, созданной в главе 4, и изменим ее так, чтобы она могла взаимодействовать с кодом на JavaScript. Добавим стандартную библиотеку C и вспомогательные функции Emscripten — это рекомендуемый способ создания модуля, предназначенного для использования в производственной среде. Позже в данной главе мы рассмотрим и другой подход к созданию модуля WebAssembly, который не включает стандартную библиотеку C или вспомогательные функции Emscripten.

Как показано на рис. 5.4, шаги по созданию модуля будут аналогичны шагам в главе 4.

1. Измените код на C++ так, чтобы он больше не получал строковый буфер и вместо этого вызывал функцию JavaScript в случае проблемы с валидацией.
2. Определите JavaScript-код, который нужно добавить в созданный Emscripten файл JavaScript.
3. Дайте Emscripten команду сгенерировать связующие файлы WebAssembly и JavaScript.
4. Скопируйте созданные файлы для использования в браузере.
5. Создайте веб-страницу, а затем напишите JavaScript-код, необходимый для взаимодействия с модулем WebAssembly.

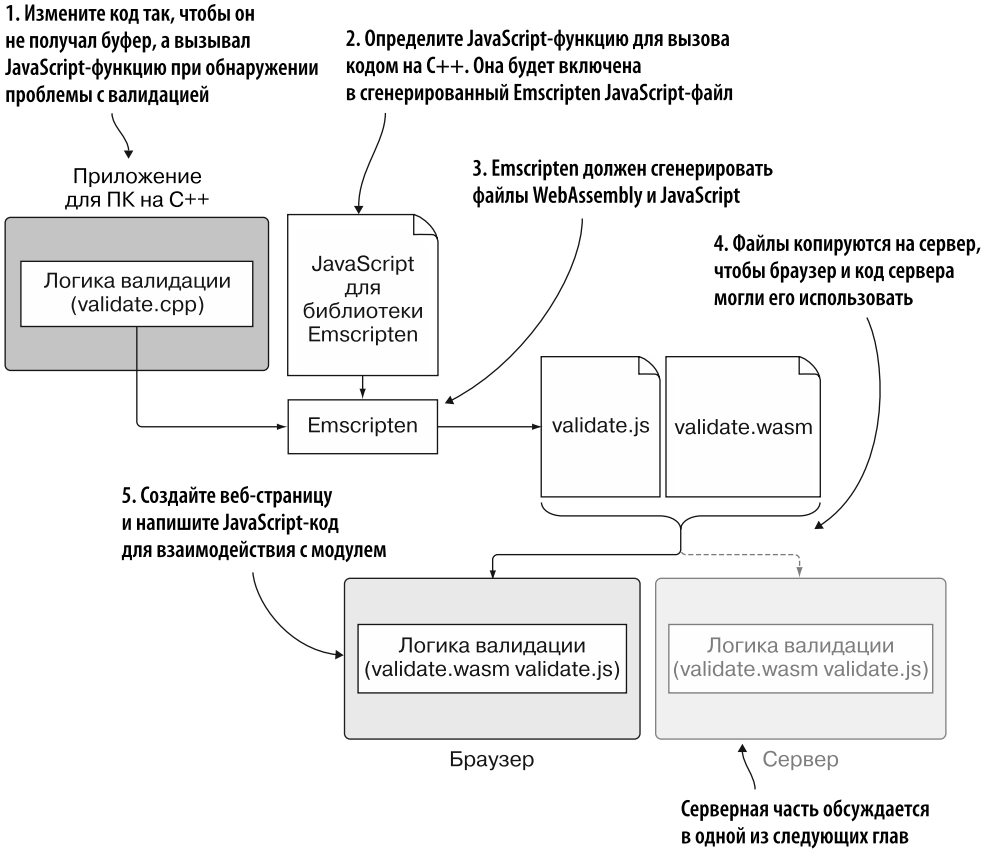


Рис. 5.4. Шаги по превращению логики на C++ и JavaScript-кода, который необходимо включить в файл JavaScript в Emscripten, в модуль WebAssembly для использования в браузере и на сервере. Серверная часть, Node.js, обсуждается в одной из следующих глав

5.1.1. Внесение изменений в код на C++

На рис. 5.5 видно, что первый шаг процесса — изменение кода на C++, чтобы он больше не получал строковый буфер. Вместо этого код вызовет функцию JavaScript, передав ей сообщение об ошибке при обнаружении проблемы с валидацией.

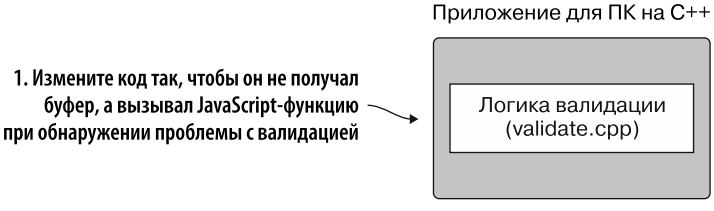


Рис. 5.5. Шаг 1 — изменение кода на C++, чтобы передать сообщение об ошибке в JavaScript-функцию

В папке WebAssembly создайте папку Chapter 5\5.1.1 EmJsLibrary\source\ для файлов, которые будут использоваться в этом подразделе. Скопируйте файл `validate.cpp` из папки WebAssembly\Chapter 4\4.1 js_plumbing\source\ в новую папку source. Откройте файл `validate.cpp` в своем любимом редакторе.

Сейчас вы измените код C++ так, чтобы вызвать функцию, определенную в JavaScript. Поскольку функция не является частью кода на C++, нужно сообщить компилятору о сигнатуре функции, добавив ключевое слово `extern` перед сигнатурой. Это позволяет компилировать код C++, ожидая, что функция будет доступна при запуске кода. Когда компилятор Emscripten видит сигнатуру функции, он создает для нее запись импорта в модуле WebAssembly. При создании экземпляра модуля фреймворк WebAssembly увидит запрошенный импорт и будет ожидать, что в JavaScript-файле будет предоставлена соответствующая функция.

Будущая функция в JavaScript принимает указатель `const char*` для параметра, который будет хранить сообщение об ошибке, если возникнет проблема с проверкой. Функция не возвращает значение. Чтобы определить сигнатуру функции, добавьте следующую строку кода в блок `extern "C"` и перед функцией `ValidateValueProvided` в файле `validate.cpp`:

```
extern void UpdateHostAboutError(const char* error_message);
```

Поскольку вы больше не собираетесь передавать буфер в модуль, необходимо удалить параметры `char* return_error_message` из функций. Кроме того, любой код, вызывающий `strcpy` для копирования сообщения об ошибке в буфер, теперь должен будет вместо этого вызвать функцию `UpdateHostAboutError`.

Измените функцию `ValidateValueProvided`, убрав параметр `return_error_message`, и добавьте вызов функции `UpdateHostAboutError`, а не `strcpy`, как показано ниже:

```
int ValidateValueProvided(const char* value,
    const char* error_message) {
  if ((value == NULL) || (value[0] == '\0')) {
    UpdateHostAboutError(error_message);
    return 0;
  }

  return 1;
}
```

Параметр return_error_message был удален

Вызов strcpy заменен вызовом UpdateHostAboutError

Подобно функции `ValidateValueProvided`, измените функцию `ValidateName`, чтобы она больше не получала параметр `return_error_message`, и удалите его из вызова функции `ValidateValueProvided`. Измените код, чтобы теперь сообщение об ошибке передавалось в функцию `UpdateHostAboutError` вместо использования `strcpy`, как показано ниже:

```
int ValidateName(char* name, int maximum_length) {
  if (ValidateValueProvided(name,
      "A Product Name must be provided.") == 0) {
    return 0;
  }

  if (strlen(name) > maximum_length) {
    UpdateHostAboutError("The Product Name is too long.");
    return 0;
  }

  return 1;
}
```

Параметр return_error_message был удален

Вызов strcpy заменен вызовом UpdateHostAboutError

В функции `IsCategoryIdInArray` никаких изменений не требуется.

Наконец, необходимо внести те же изменения, что и для функций `ValidateValueProvided` и `ValidateName`, в функцию `ValidateCategory`, как показано в листинге 5.1.

Листинг 5.1. Измененная функция ValidateCategory в файле validate.cpp

```
int ValidateCategory(char* category_id, int* valid_category_ids,
    int array_length) {
  if (ValidateValueProvided(category_id,
      "A Product Category must be selected.") == 0) {
    return 0;
  }
```

Параметр return_error_message был удален

```
  if ((valid_category_ids == NULL) || (array_length == 0)) {
    UpdateHostAboutError("There are no Product Categories available.");
    return 0;
  }

  if (IsCategoryIdInArray(category_id, valid_category_ids,
      array_length) == 0) {
    UpdateHostAboutError("The selected Product Category is not valid.");
    return 0;
  }

  return 1;
}
```

5.1.2. Создание кода на JavaScript и добавление его в сгенерированный Emscripten JavaScript-файл

После того как код на C++ будет изменен, можно сделать следующий шаг — создать код на JavaScript и добавить его в JavaScript-файл, созданный Emscripten (рис. 5.6).

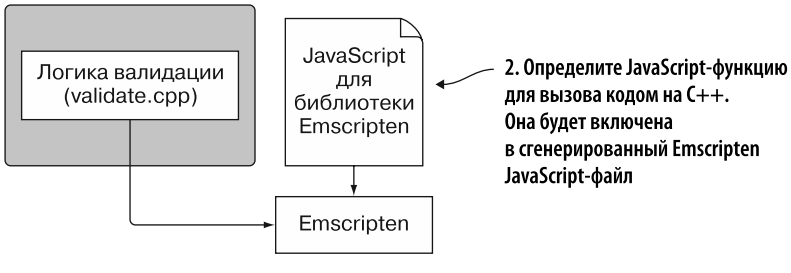


Рис. 5.6. Шаг 2 — создание кода на JavaScript и добавление его в JavaScript-файл, сгенерированный Emscripten

При написании кода на JavaScript, который будет включен в созданный Emscripten JavaScript-файл, модуль WebAssembly создается немного другим способом. В этом случае вы определите свою функцию `UpdateHostAboutError` в JavaScript до того, как дадите Emscripten указание скомпилировать код C++, поскольку с помощью компилятора Emscripten вы можете объединить ваш JavaScript-код с остальной частью JavaScript-кода, генерируемого Emscripten.

Чтобы ваш JavaScript был включен в созданный Emscripten JavaScript-файл, необходимо добавить его в объект `LibraryManager.library` в Emscripten; для этого можно использовать функцию `mergeInto` в Emscripten, которая принимает два параметра:

- объект, к которому нужно добавить свойства, — в данном случае объект `LibraryManager.library`;
- объект, свойства которого будут скопированы в первый объект, — в нашем случае ваш код на JavaScript.

Вы создадите JavaScript-объект, который будет содержать свойство `UpdateHostAboutError`; значение его — функция, получающая указатель на сообщение об ошибке. Функция считывает строку из памяти модуля с помощью вспомогательной функции в Emscripten — `UTF8ToString` — и затем вызывает JavaScript-функцию `setErrorMessage`, которая является частью основного кода JavaScript вашей веб-страницы.

В папке WebAssembly\Chapter 5\5.1.1 EmJsLibrary\source\ создайте файл `mergeinto.js`, откройте его в своем любимом редакторе и добавьте следующий фрагмент кода:

```
mergeInto(LibraryManager.library, {
  UpdateHostAboutError: function(errorMessagePointer) {
    setErrorMessage(Module.UTF8ToString(errorMessagePointer));
  }
});
```

Копирует свойства объекта в объект LibraryManager.library

5.1.3. Компиляция кода в модуль WebAssembly

Изменив код на C++ и создав функцию JavaScript, которую нужно включить в JavaScript-файл, созданный Emscripten, можно сделать следующий шаг. Как показано на рис. 5.7, на данном этапе Emscripten скомпилирует код в модуль WebAssembly. Кроме того, Emscripten получит указание добавить код из вашего файла `mergeinto.js` в сгенерированный файл JavaScript.

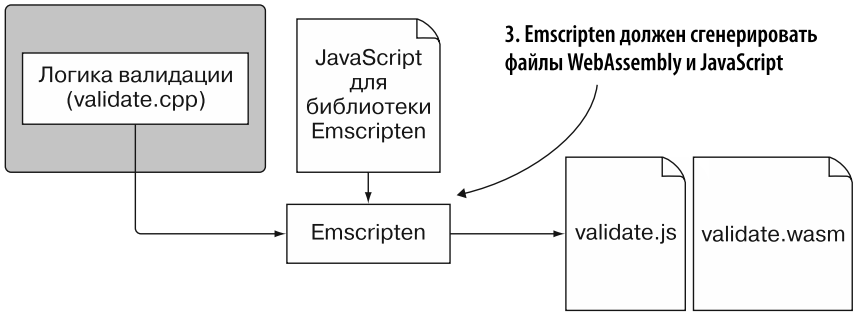


Рис. 5.7. Шаг 3 — даем Emscripten команду сгенерировать файлы WebAssembly и JavaScript. В нашем случае мы дадим Emscripten указание включить файл mergeInto.js

Чтобы дать компилятору Emscripten указание включить ваш код на JavaScript в сгенерированный JavaScript-файл, нужно использовать флаг `--js-library`, за которым следует путь к добавляемому файлу. Чтобы быть уверенными в том, что вспомогательные функции Emscripten, необходимые вашему коду на JavaScript, включены в сгенерированный JavaScript-файл, вы должны указать их при компиляции кода на C++, добавив их в массив параметра `EXTRA_EXPORTED_RUNTIME_METHODS`. Укажите следующие вспомогательные функции Emscripten:

- `ccall` — используется JavaScript-кодом веб-страницы для вызова модуля;
- `UTF8ToString` — применяется JavaScript-кодом, который вы написали в файле `mergeinto.js`, для чтения строк из памяти модуля.

Чтобы скомпилировать код в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой вы сохранили файлы `validate.cpp` и `mergeinto.js`, и выполните следующую команду:

```
emcc validate.cpp --js-library mergeinto.js
➥ -s EXTRA_EXPORTED_RUNTIME_METHODS=['ccall','UTF8ToString']
➥ -o validate.js
```

Если вы откроете сгенерированный Emscripten файл JavaScript, `validate.js`, и выполните поиск функции `UpdateHostAboutError`, то увидите, что функция, которую вы сейчас определили, является частью сгенерированного JavaScript-файла:

```
function _UpdateHostAboutError(errorMessagePointer) {
  setErrorMessage(Module.UTF8ToString(errorMessagePointer));
}
```

Одной из приятных особенностей добавления функций в сгенерированный файл JavaScript является то, что если, помимо `UpdateHostAboutError`, в файле есть несколько других функций, то будут включены только те, которые фактически вызываются кодом модуля.

5.1.4. Изменение JavaScript-кода веб-страницы

На рис. 5.8 показан следующий шаг процесса — копирование файлов, созданных Emscripten, в папку с копиями файлов `editproduct.html` и `editproduct.js`, созданных в главе 4. Затем мы изменим часть кода в файле `editproduct.js`, исходя из предполагаемого взаимодействия с модулем.

В папке WebAssembly\Chapter 5\5.1.1 EmJsLibrary\ создайте папку frontend. Скопируйте в нее следующие файлы:

- `validate.js` и `validate.wasm` из папки Chapter 5\5.1.1 EmJsLibrary\source;
- `editproduct.html` и `editproduct.js` из папки Chapter 4\4.1 js_plumbing\frontend\.

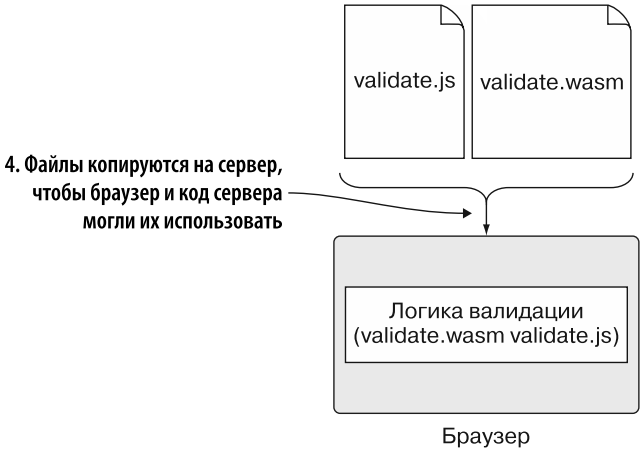


Рис. 5.8. Шаг 4 — копирование сгенерированных файлов в папку с файлом HTML и обновление JavaScript-кода исходя из того, как он будет взаимодействовать с модулем

Откройте файл `editproduct.js` в редакторе.

JavaScript больше не нуждается в создании строкового буфера и передаче его в модуль, поэтому можно упростить функцию `onClickSave` в файле `editproduct.js`.

- Переменные `errorMessage` и `errorMessagePointer` больше не нужны, так что можете удалить эти две строки кода. Вместо них вы вызовете функцию `setErrorMessage` и передадите пустую строку — если на веб-странице отображалась предыдущая ошибка, то при отсутствии проблем с валидацией во время текущего вызова функции сохранения сообщение будет скрыто.
- Удалите параметр `errorMessagePointer` из вызовов функций `validateName` и `validateCategory`.
- Удалите строку кода `Module.UTF8ToString` из оператора `if`.
- Измените оператор `if` так, чтобы условие `ИЛИ` (`||`) между двумя проверками теперь было условием `И` (`&&`), и удалите проверку неравенства (`!`) перед обоими вызовами функций. Теперь, если оба вызова функций указывают на отсутствие ошибок, значит, все в порядке и данные можно передать в код на стороне сервера.

- Можно удалить остальную часть кода, следующего за оператором `if` в функции.

Функция `onClickSave` теперь должна выглядеть так:

```
function onClickSave() {
  setErrorMessage("");

  const name = document.getElementById("name").value;
  const categoryId = getSelectedCategoryId();

  if (validateName(name) &&
      validateCategory(categoryId)) {

  }
}
```

Очищает предыдущее сообщение об ошибке

Второй параметр каждого вызова функции был удален

Проблем не обнаружено. Данные могут быть переданы в код серверной стороны

Проверки неравенства перед вызовами функций были удалены. Условие ИЛИ заменено на И

Вам также потребуется изменить функцию `validateName`:

- удалите параметр `errorMessagePointer`;
- поскольку функция `ValidateName` в модуле WebAssembly теперь ожидает только два параметра, удалите последний элемент массива `('number')` в третьем параметре функции `ccall`;
- удалите элемент массива `errorMessagePointer` из последнего параметра функции `ccall`.

Функция `validateName` теперь должна выглядеть следующим образом:

```
function validateName(name) {
  const isValid = Module.ccall('ValidateName',
      'number',
      ['string', 'number'],
      [name, MAXIMUM_NAME_LENGTH]);

  return (isValid === 1);
}
```

Второй параметр (errorMessagePointer) был удален

Третий элемент массива (number) был удален

Третий элемент массива (errorMessagePointer) был удален

Внесите те же изменения, что и в функции `validateName`, в функцию `validateCategory`:

- удалите параметр `errorMessagePointer`;
- удалите последний элемент массива `('number')` из третьего параметра функции `ccall`;
- удалите элемент массива `errorMessagePointer` из последнего параметра функции `ccall`.

Функция `validateCategory` теперь должна выглядеть так, как показано в листинге 5.2.

Листинг 5.2. Измененная функция validateCategory в файле editproduct.js

```
function validateCategory(categoryId) {
  const arrayLength = VALID_CATEGORY_IDS.length;
  const bytesPerElement = Module.HEAP32.BYTES_PER_ELEMENT;
  const arrayPointer = Module._malloc((arrayLength * bytesPerElement));
  Module.HEAP32.set(VALID_CATEGORY_IDS, (arrayPointer / bytesPerElement));

  const isValid = Module.ccall('ValidateCategory',
      'number',
      ['string', 'number', 'number'],
      [categoryId, arrayPointer, arrayLength]);

  Module._free(arrayPointer);

  return (isValid === 1);}
```

Второй параметр (errorMessagePointer) был удален

Четвертый элемент массива (number) был удален

Четвертый элемент массива (errorMessagePointer) был удален

5.1.5. Просмотр результатов

Закончив вносить изменения в JavaScript-код, вы можете открыть браузер и ввести `http://localhost:8080/editproduct.html` в адресную строку, чтобы увидеть получившуюся веб-страницу. Протестируйте валидацию, удалив весь текст из поля Name (Название) и нажав кнопку Save (Сохранить). На веб-странице должна отображаться ошибка (рис. 5.9).

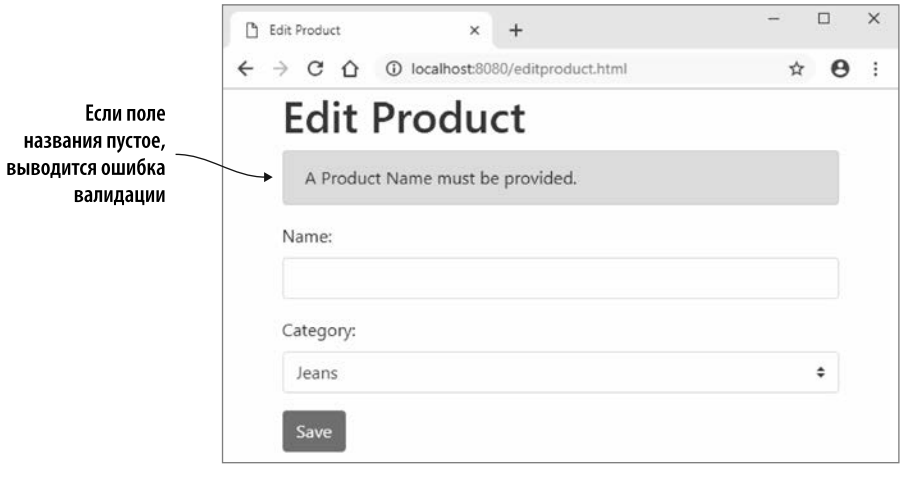


Рис. 5.9. Ошибка валидации при изменении названия на странице Edit Product (Изменить товар)

5.2. ИСПОЛЬЗОВАНИЕ C ИЛИ C++ ДЛЯ СОЗДАНИЯ МОДУЛЯ БЕЗ СВЯЗУЮЩИХ ФАЙЛОВ EMSCRIPTEN

Предположим, вы хотите, чтобы Emscripten скомпилировал код на C++ и при этом не включил какие-либо стандартные функции библиотеки C и не сгенерировал связующий файл JavaScript. Связующий код Emscripten удобен и вдобавок скрывает многие детали работы с модулем WebAssembly. Подход, который вы здесь увидите, очень полезен для обучения, поскольку вы будете работать с модулем напрямую.

Процесс, представленный в разделе 5.1 со связующим кодом Emscripten, обычно используется для производственного кода. Сгенерированный Emscripten файл JavaScript удобен тем, что выполняет загрузку и создание экземпляра модуля, а также включает вспомогательные функции, упрощающие взаимодействие с модулем.

В разделе 5.1, когда вы скомпилировали модуль WebAssembly и добавили связующий код Emscripten, функция `updateHostAboutError` была помещена в созданный Emscripten файл JavaScript, как показано на рис. 5.10.

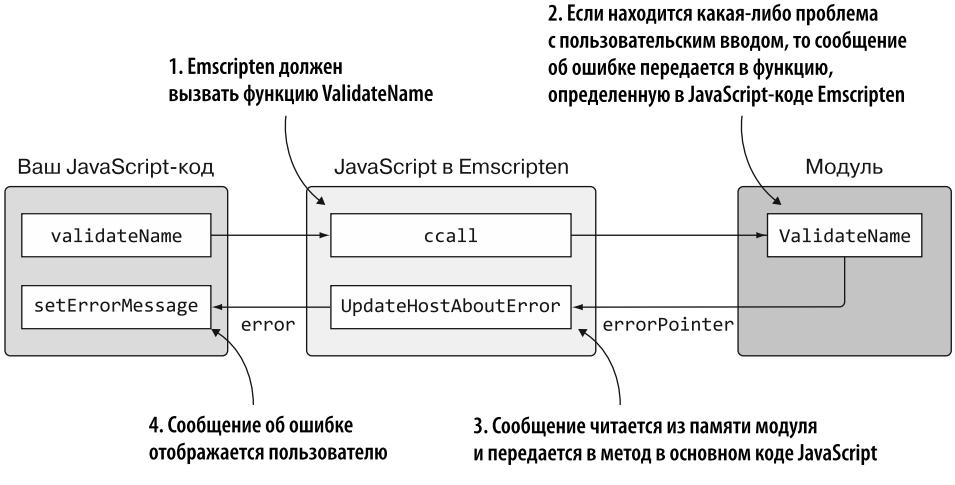


Рис. 5.10. Вызов модулем JavaScript-кода через функцию, определенную в файле JavaScript, сгенерированном Emscripten

Если связующий код Emscripten не используется, то ваш код на C или C++ не будет иметь доступа к макросам Emscripten или файлу JavaScript Emscripten, но по-прежнему может напрямую вызывать JavaScript. У вас не будет доступа к созданным Emscripten файлам JavaScript, поэтому функцию обратного вызова необходимо поместить в файл JavaScript сайта, как показано на рис. 5.11.

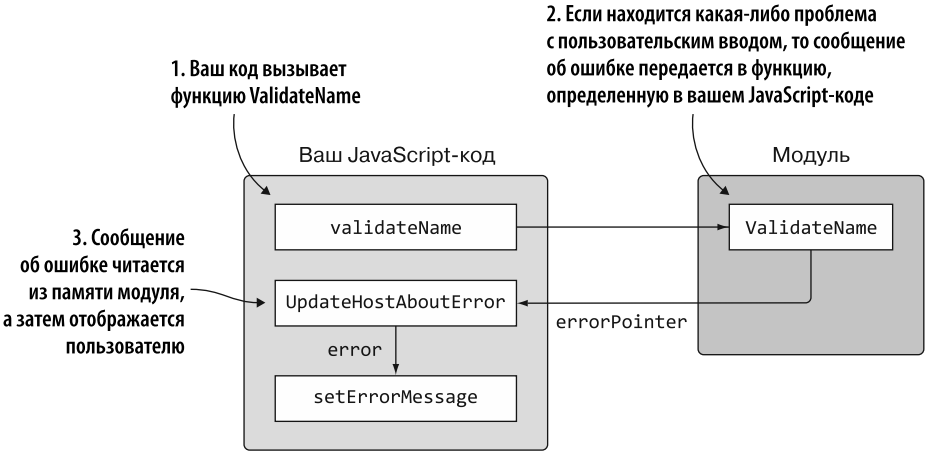


Рис. 5.11. Логика обратных вызовов без использования связующих файлов Emscripten

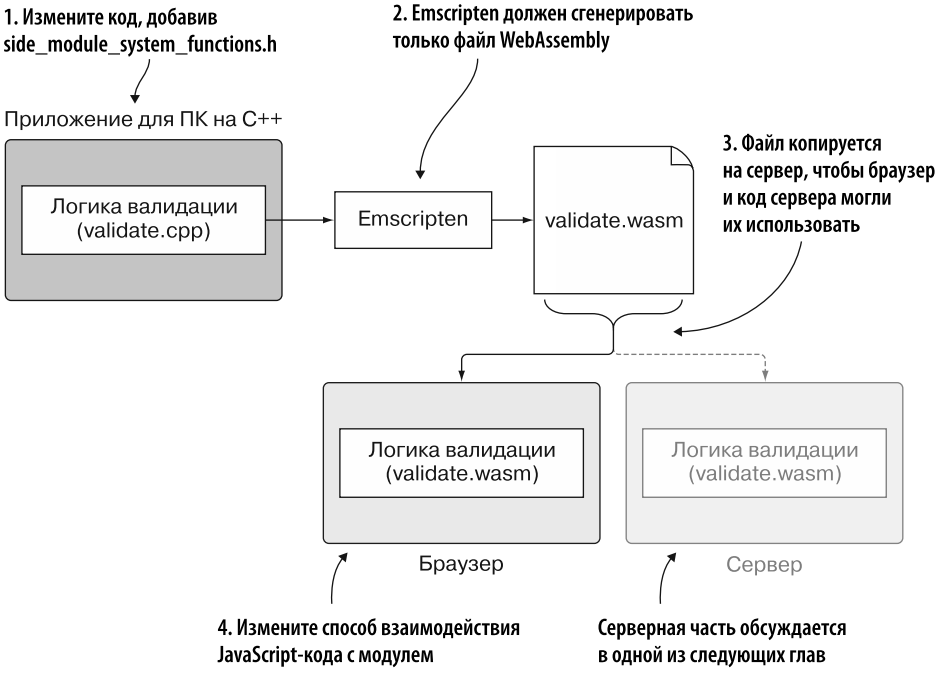


Рис. 5.12. Процесс, при котором существующая логика на C++ превращается в модуль WebAssembly для использования кодом сайта и сервера — но без генерации JavaScript-кода Emscripten. Серверная часть, Node.js, обсуждается в одной из следующих глав

В подразделе 5.1.1 я предупреждал вас, что при добавлении JavaScript в код JavaScript в Emscripten код должен быть автономным, если вы планируете использовать Node.js. В главе 10 вы будете работать с модулями WebAssembly в Node.js и рассмотрите процесс более подробно, но мое предупреждение связано с тем, как файлы JavaScript, созданные Emscripten, загружаются в Node.js.

Модули, созданные с помощью этого подхода, не имеют ограничений автономного кода — код, вызываемый модулем, будет частью вашего основного JavaScript-кода. Как вы можете видеть на рис. 5.12, процесс аналогичен описанному в разделе 5.1, за исключением того, что вы дадите Emscripten указание сгенерировать только файл WebAssembly.

5.2.1. Внесение изменений в код на C++

Первый шаг процесса (рис. 5.13) — изменение кода на C++, созданного в разделе 5.1, так, чтобы он использовал файлы `side_module_system_functions.h` и `.cpp`, созданные в главе 4. В папке Chapter 5\ создайте папку 5.2.1 SideModuleCallingJS\source\ для ваших файлов в этом подразделе. Скопируйте в новую папку source следующие файлы:

- `validate.cpp` из папки 5.1.1 EmJsLibrary\source\;
- `side_module_system_functions.h` и `.cpp` из папки Chapter 4\4.2 side_module\source\.

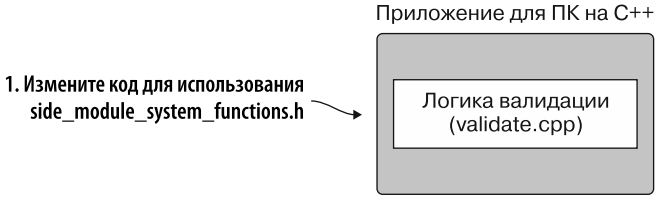


Рис. 5.13. Нужно изменить код на C++ из раздела 5.1 так, чтобы он использовал файлы side_module_system_functions, созданные в главе 4

Когда дело доходит до прямого вызова JavaScript, мы используем код на C++, идентичный созданному в разделе 5.1, в котором ключевое слово `extern` служит для определения сигнатуры функции JavaScript:

```
extern void UpdateHostAboutError(const char* error_message);
```

Единственное различие между кодом на C++ здесь и кодом, написанным в разделе 5.1, состоит в том, что этот код не будет иметь доступа к стандартной библиотеке C. Вам нужно будет импортировать код, добавленный в главе 4, — он предоставляет такие функции, как `strcpy`, `strlen` и `atoi`.

Откройте файл `validate.cpp` в своем любимом редакторе, а затем удалите строки импорта для стандартной системной библиотеки cstdlib и cstring. Затем добавьте заголовочный файл для вашей версии стандартных функций библиотеки C, `side_module_system_functions.h`, в блок `extern "C"`.

ПРЕДУПРЕЖДЕНИЕ

Подключаемый заголовочный файл должен быть помещен в блок extern "C", потому что компилятору Emscripten нужно будет скомпилировать два файла .cpp. Хотя функции обоих файлов находятся внутри блоков extern "C", компилятор Emscripten по-прежнему предполагает, что вызовы функций в файле validate.cpp компилируются в файл C++ с искаженными именами функций. Компилятор не увидит искаженные имена функций в сгенерированном модуле и будет считать, что вместо этого их нужно импортировать.

В следующем фрагменте кода показаны изменения в файле `validate.cpp`:

```
#ifdef __EMSCRIPTEN__
  #include <emscripten.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif

#include "side_module_system_functions.h"
```

Важно: поместите заголовочный файл в блок extern "C"

5.2.2. Компиляция кода в модуль WebAssembly

После того как код на C++ изменен, следующий шаг — компиляция его с помощью Emscripten в модуль WebAssembly, но без связующего JavaScript-кода, как показано на рис. 5.14.

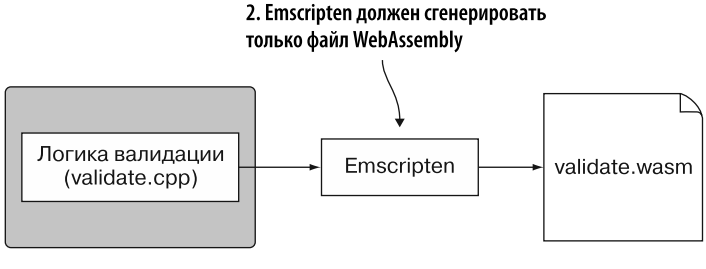


Рис. 5.14. В данном случае Emscripten должен сгенерировать только файл WebAssembly, без связующего файла JavaScript

Чтобы скомпилировать код на C++ в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой хранятся файлы на C++, и выполните следующую команду:

```
emcc side_module_system_functions.cpp validate.cpp
➥ -s SIDE_MODULE=2 -O1 -o validate.wasm
```

5.2.3. Изменение кода на JavaScript для взаимодействия с модулем

Модуль WebAssembly сгенерирован. На рис. 5.15 показан следующий шаг: копирование сгенерированного файла Wasm в папку, в которой находится файл HTML. Затем вы измените способ взаимодействия JavaScript-кода с модулем, учитывая, что он не передает буфер в функции модуля.

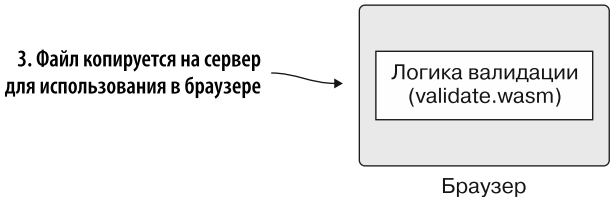


Рис. 5.15. Нужно скопировать сгенерированный файл Wasm в папку, в которой находится файл HTML, и изменить способ взаимодействия JavaScript-кода с модулем

В папке Chapter 5\5.2.1 SideModuleCallingJS\ создайте папку frontend\. Скопируйте в нее следующие файлы:

- только что созданный файл `validate.wasm` из папки 5.2.1 SideModuleCallingJS\source\;
- файлы `editproduct.html` и `editproduct.js` из папки Chapter 4\4.2 side_module\frontend\.

В коде на C++ ключевое слово `extern` и сигнатура функции сообщают компилятору Emscripten, что модуль будет импортировать функцию `_UpdateHostAboutError` (компилятор Emscripten добавляет подчеркивание перед именем функции в сгенерированном модуле WebAssembly). Связующий код Emscripten не используется, когда JavaScript создает экземпляр модуля, поэтому вы сами должны передать ему функцию `_UpdateHostAboutError`.

Функция initializePage

Первый шаг — открыть файл `editproduct.js` в редакторе, а затем найти функцию `initializePage`. Измените `importObject`, добавив в конец новое свойство `_UpdateHostAboutError` и функцию, которая получает параметр `errorMessagePointer`. Внутри тела функции вы вызовете функцию `getStringFromMemory` для чтения строки из памяти модуля. Затем передадите строку в функцию `setErrorMessage`.

В листинге 5.3 показано, как теперь должен выглядеть `importObject` в функции `initializePage` файла `editproduct.js`.

Листинг 5.3. В importObject добавлена функция _UpdateHostAboutError

```
function initializePage() {
  ...

  moduleMemory = new WebAssembly.Memory({initial: 256});

  const importObject = {
    env: {
      __memory_base: 0,
      memory: moduleMemory,
      _UpdateHostAboutError: function(errorMessagePointer) {
        setErrorMessage(getStringFromMemory(errorMessagePointer));
      },
    }
  };

  ...
}
```

Функция, созданная для ответа на вызовы из модуля

Читает строку из памяти модуля и отображает ее пользователю

Остальные изменения в файле `editproduct.js` будут такими же, как и в разделе 5.1: удаление переменной буфера ошибок из функций `onClickSave`, `validateName` и `validateCategory`.

Функция onClickSave

Найдите функцию `onClickSave` и выполните следующие действия.

- Замените строки `errorMessage` и `errorMessagePointer` вызовом `setErrorMessage` с передачей пустой строки. Если нет проблем при валидации, то вызов функции `setErrorMessage` с пустой строкой удалит все сообщения об ошибках, которые могли отображаться в последний раз, когда пользователь нажимал кнопку Save (Сохранить).
- Измените оператор `if`, чтобы он больше не передавал параметр `errorMessagePointer`.
- Удалите проверки неравенства (`!`) перед вызовами функций `validateName` и `validateCategory`. Измените условие ИЛИ (`||`) на условие И (`&&`).
- Удалите строку `getStringFromMemory` из тела оператора `if`. Это условие проверяет, что валидация прошла успешно, и в тело `if` вы поместите код для передачи данных на сервер для сохранения.
- Удалите остальную часть кода, следующего за оператором `if` в функции `onClickSave`.

Функция `onClickSave` теперь должна выглядеть так, как показано во фрагменте кода ниже:

```
function onClickSave() {
  setErrorMessage("");

  const name = document.getElementById("name").value;
  const categoryId = getSelectedCategoryId();

  if (validateName(name) &&
      validateCategory(categoryId)) {

  }
}
```

Очищает предыдущее сообщение об ошибке

Второй параметр каждого вызова функции был удален

Проверки неравенства удалены. Условие ИЛИ заменено на И

Проблем не обнаружено. Данные могут быть переданы в код серверной стороны

Функции validateName и validateCategory

Следующий шаг — изменение функций `validateName` и `validateCategory`, чтобы они больше не получали параметр `errorMessagePointer` и не передавали значение функциям модуля. В листинге 5.4 показаны измененные функции.

Листинг 5.4. Изменение функций validateName и validateCategory

```
function validateName(name) {
  const namePointer = moduleExports._create_buffer((name.length + 1));
  copyStringToMemory(name, namePointer);

  const isValid = moduleExports._ValidateName(namePointer,
      MAXIMUM_NAME_LENGTH);

  moduleExports._free_buffer(namePointer);

  return (isValid === 1);
}

function validateCategory(categoryId) {
  const categoryIdPointer = moduleExports._create_buffer(
➥ (categoryId.length + 1));
  copyStringToMemory(categoryId, categoryIdPointer);

  const arrayLength = VALID_CATEGORY_IDS.length;
  const bytesPerElement = Int32Array.BYTES_PER_ELEMENT;
  const arrayPointer = moduleExports._create_buffer((arrayLength *
➥ bytesPerElement));

  const bytesForArray = new Int32Array(moduleMemory.buffer);
  bytesForArray.set(VALID_CATEGORY_IDS, (arrayPointer / bytesPerElement));

  const isValid = moduleExports._ValidateCategory(categoryIdPointer,
```

Второй параметр функции, errorMessagePointer, удален

errorMessagePointer больше не передается в функцию модуля

Второй параметр функции, errorMessagePointer, удален

```
      arrayPointer, arrayLength);

  moduleExports._free_buffer(arrayPointer);
  moduleExports._free_buffer(categoryIdPointer);

  return (isValid === 1);
}
```

errorMessagePointer больше не передается в функцию модуля

5.2.4. Просмотр результатов

Теперь, когда вы все настроили, введите `http://localhost:8080/editproduct.html` в адресную строку браузера, чтобы увидеть получившуюся веб-страницу. Вы можете проверить правильность работы валидации, изменив выбор в раскрывающемся списке Category (Категория) так, чтобы ничего не было выбрано, а затем нажав кнопку Save (Сохранить). Логика валидации должна отобразить сообщение об ошибке на веб-странице, как показано на рис. 5.16.

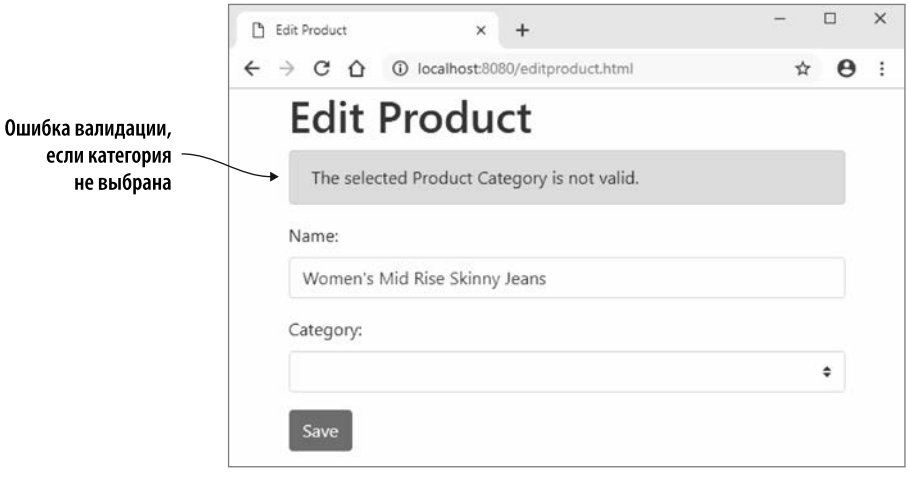


Рис. 5.16. Ошибка валидации на странице Edit Product (Изменить товар), если категория не выбрана

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Благодаря возможности вызывать JavaScript ваш модуль теперь может взаимодействовать с веб-страницей и веб-API браузера, что открывает множество возможностей. Ниже представлены лишь некоторые из них.

- Создание модуля WebAssembly, который выполняет вычисления по трассировке лучей для 3D-графики. Затем эту графику можно будет применять для интерактивной веб-страницы или игры.
- Создание конвертера файлов (например, выбрать снимок и преобразовать его в PDF перед прикреплением к электронному письму).
- Взять существующую библиотеку C++ с открытым исходным кодом — например, библиотеку криптографии — и скомпилировать ее в WebAssembly для использования на вашем сайте. На сайте https://en.cppreference.com/w/cpp/links/libs перечислены библиотеки C++ с открытым исходным кодом.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какое ключевое слово нужно использовать для определения сигнатуры в вашем коде на C или C++, чтобы компилятор знал, что функция будет доступна при запуске кода?
2. Предположим, вам нужно добавить функцию в код JavaScript в Emscripten, которую ваш модуль будет вызывать, чтобы определять, подключено ли устройство пользователя к сети. Как бы вы добавили функцию `IsOnline`, которая возвращает `1`, если истина, и `0` (ноль), если ложь?

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Можно изменить модуль WebAssembly так, чтобы он мог напрямую взаимодействовать с JavaScript-кодом.
- Внешние функции могут быть определены в вашем коде на C или C++ с помощью ключевого слова `extern`.
- Можно добавить собственный код JavaScript в JavaScript-файл, созданный Emscripten, добавив его в объект `LibraryManager.library`.
- Если связующий код Emscripten не используется, то можно включить функцию для импортирования модуля, поместив ее в объект JavaScript, который вы передаете функциям `WebAssembly.instantiate` или `WebAssembly.instantiateStreaming`.

6 Создание модуля WebAssembly, вызывающего JavaScript, с использованием указателей на функции

В этой главе

- ✓ Изменение кода на C или C++ для работы с указателями на функции.
- ✓ Применение вспомогательных функций Emscripten для передачи функций JavaScript в модуль WebAssembly.
- ✓ Вызов указателей на функции в модуле WebAssembly, если код Emscripten не используется.

В главе 5 вы изменили модуль так, чтобы он больше не передавал сообщение об ошибке валидации обратно в JavaScript через параметр. Вместо этого вы изменили модуль так, чтобы он напрямую вызывал JavaScript-функцию, как показано на рис. 6.1.

Представьте, что вы можете передать функцию JavaScript в модуль в зависимости от текущих потребностей вашего JavaScript-кода. Завершая обработку, модуль может вызвать указанную функцию, как показано на рис. 6.2.

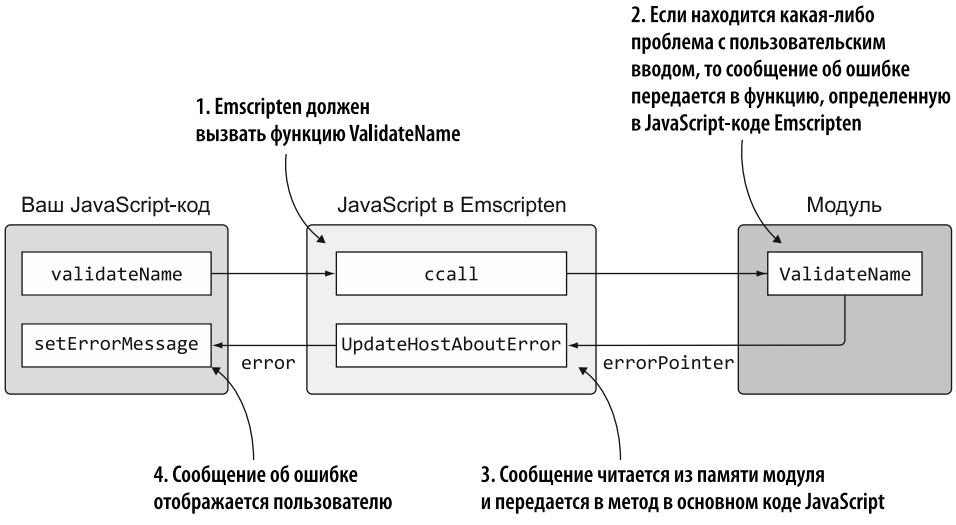


Рис. 6.1. Вызов модулем функции в JavaScript-коде

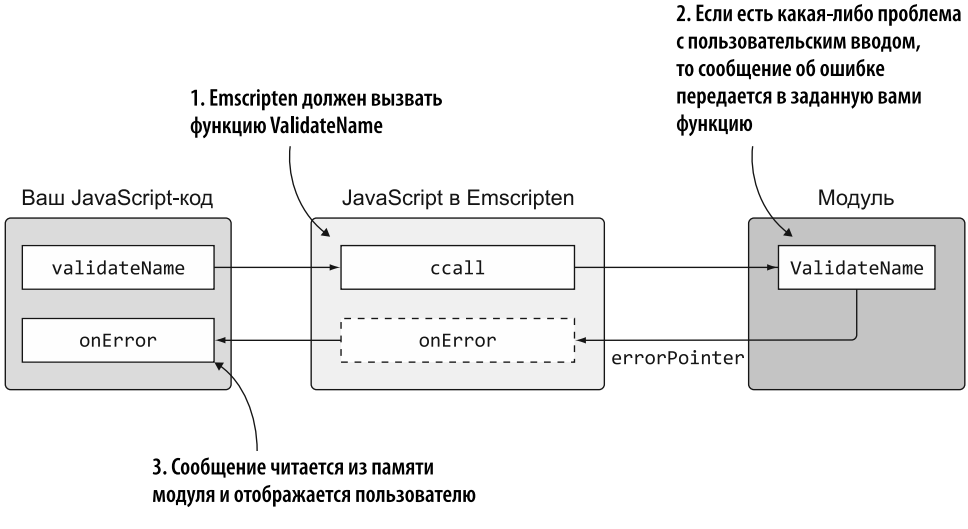


Рис. 6.2. Модуль, вызывающий JavaScript через указатель на функцию

6.1. ИСПОЛЬЗОВАНИЕ C ИЛИ C++ ДЛЯ СОЗДАНИЯ МОДУЛЯ СО СВЯЗУЮЩИМИ ФАЙЛАМИ EMSCRIPTEN

В этом разделе мы создадим код на C++ для логики валидации. Добавим стандартную библиотеку C и вспомогательные функции Emscripten — это рекомендуемый способ создания модуля для использования в производственной среде. Позже в данной главе вы познакомитесь с другим подходом к созданию модуля WebAssembly, который не включает стандартную библиотеку C или вспомогательные функции Emscripten.

6.1.1. Использование указателя на функцию, передаваемого модулю через JavaScript

Как показано на рис. 6.3, настройка модуля таким образом, чтобы он использовал указатели на функции, потребует выполнения следующих шагов.

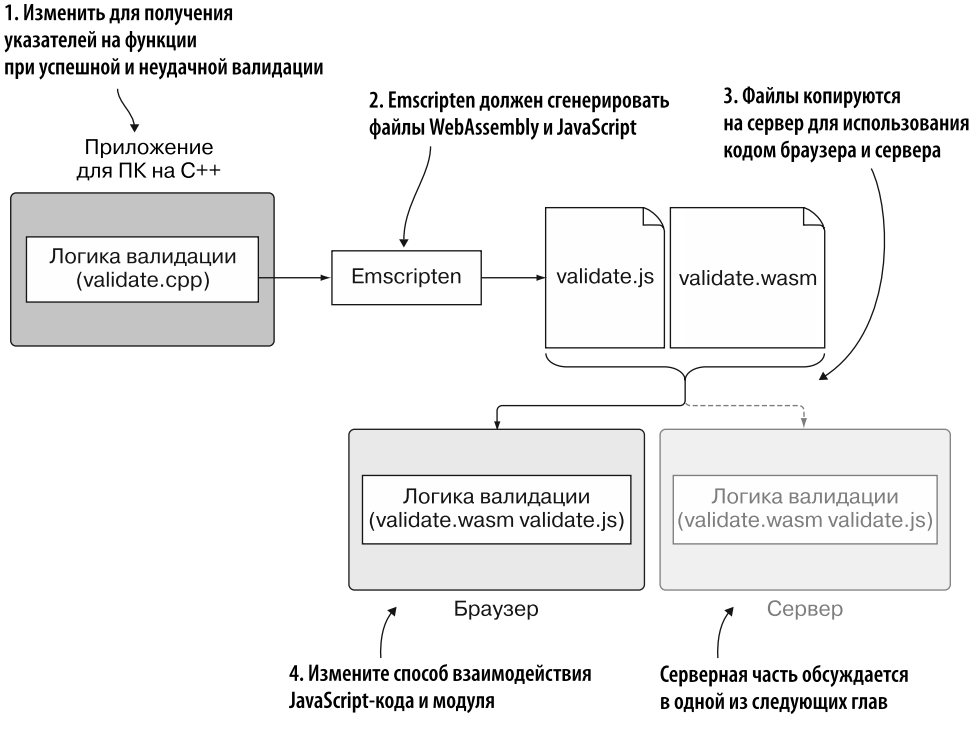


Рис. 6.3. Изменение существующей логики на C++, чтобы код мог принимать указатели на функции и его можно было собрать в WebAssembly для использования в коде сайта и сервера. Серверная часть, Node.js, обсуждается в одной из следующих глав

1. Измените код на C++ так, чтобы экспортируемые функции получали указатели на функции, вызываемые при успешной и неудачной валидации.
2. Дайте Emscripten команду сгенерировать файл WebAssembly и связующий файл JavaScript.
3. Скопируйте сгенерированные файлы для использования в браузере.
4. Измените JavaScript-код сайта, чтобы он взаимодействовал с модулем WebAssembly, ожидая передачи указателей на функции.

6.1.2. Изменение кода на C++

Первым делом необходимо изменить код на C++, чтобы принимать указатели на функции, как показано на рис. 6.4.

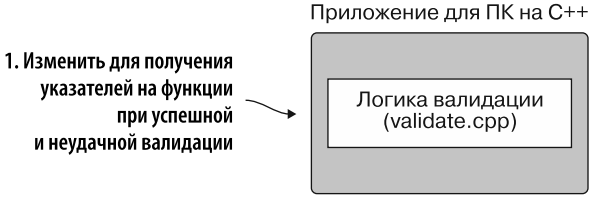


Рис. 6.4. Шаг 1 — изменение кода так, чтобы он принимал указатели на функции

Создайте следующую папку для хранения файлов этого подраздела: WebAssembly\Chapter 6\6.1.2 EmFunctionPointers\source\. Скопируйте файл `validate.cpp` из папки WebAssembly\Chapter 5\5.1.1 EmJsLibrary\source\ в только что созданную папку source. Затем откройте его в своем любимом редакторе, чтобы определить сигнатуры функций, которые ваш код будет использовать для вызова кода JavaScript в целях определения либо успеха, либо наличия проблемы с пользовательскими данными.

Определение сигнатур функций

В C или C++ функции могут принимать параметр с сигнатурой указателя на функцию. Например, следующий параметр — указатель на функцию, которая не принимает никаких параметров и не возвращает значение:

```
void(*UpdateHostOnSuccess)(void)
```

Вы можете встретить примеры кода, в которых указатель на функцию вызывается с предварительным разыменованием указателя. В этом нет необходимости, поскольку указатель на разыменованную функцию немедленно преобразуется

в обычный указатель, поэтому вы просто получаете тот же указатель на функцию. Код на C может вызывать указатель на функцию таким же образом, что и обычную функцию, как показано в следующем примере:

```
void Test(void(*UpdateHostOnSuccess)(void)) {
  UpdateHostOnSuccess();
}
```

Хотя вы можете указать сигнатуру функции в качестве параметра в каждой функции, где это необходимо, также можно создать определение данной сигнатуры и вместо этого использовать его для параметров. Определение сигнатуры функции создается с помощью ключевого слова `typedef`, за которым следует сигнатура.

Использование предопределенной сигнатуры функции вместо определения сигнатуры функции для каждого параметра имеет некоторые преимущества:

- упрощаются функции;
- повышается удобство сопровождения. Если вам когда-либо понадобится изменить сигнатуру функции, то не придется изменять каждый параметр, в котором она используется. Вместо этого достаточно обновить только одно место в коде — определение.

Вы будете использовать подход с `typedef` для определения двух сигнатур функций, необходимых коду в файле `validate.cpp`:

- одна сигнатура — для функции успешного обратного вызова, которая не будет иметь никаких параметров и не станет возвращать значение;
- вторая сигнатура — для функции обратного вызова с ошибкой валидации. Она принимает параметр `const char*` и не возвращает значение.

В файле `validate.cpp` замените строку `extern void UpdateHostAboutError` двумя сигнатурами, показанными в следующем фрагменте кода:

```
typedef void(*OnSuccess)(void);
typedef void(*OnError)(const char*);
```

Теперь, когда модуль не будет получать параметр буфера для возврата сообщения об ошибке, удалим этот параметр из функций модуля, начиная с функции `ValidateValueProvided`.

Функция ValidateValueProvided

Измените функцию `ValidateValueProvided`, удалив параметр `error_message`. Затем удалите вызов `UpdateHostAboutError` из оператора `if`. Измененная функция `ValidateValueProvided` теперь должна выглядеть следующим образом:

```
int ValidateValueProvided(const char* value) {
  if ((value == NULL) || (value[0] == '\0')) {
    return 0;
  }

  return 1;
}
```

Параметр error_message был удален

Код больше не вызывает UpdateHostAboutError

Затем нужно изменить функции `ValidateName` и `ValidateCategory`, чтобы они получали указатели на функции успеха и ошибки для вызова соответствующей функции в зависимости от наличия проблем с пользовательскими данными.

Функция ValidateName

Необходимо внести несколько изменений в функцию `ValidateName`. Начните с замены типа возвращаемого значения функции с `int` на `void`, а затем добавьте два параметра указателя функции:

- `OnSuccess UpdateHostOnSuccess`;
- `OnError UpdateHostOnError`.

Поскольку вы удалили второй параметр из функции `ValidateValueProvided`, теперь в нее нельзя передать строку, поэтому удалите второй параметр из вызова функции. Замените строку `return 0` внутри данного оператора `if` на вызов указателя функции ошибки:

```
UpdateHostOnError("A Product Name must be provided.");
```

Первоначально функция JavaScript, которую вызывал код, называлась `UpdateHostAboutError`. Вы ее удалили, и теперь нам нужно, чтобы код для условия `if` с проверкой длины строки (`strlen`) вызывал указатель на функцию ошибки. Переименуйте вызов функции `UpdateHostAboutError` на `UpdateHostOnError`, а затем удалите строку `return 0`.

Функция `ValidateName` теперь возвращает `void`, поэтому необходимо удалить строку `return 1` из конца функции и заменить ее на оператор `else` в конце блока `if`. Блок `else` запускается, если проблем с пользовательским вводом не обнаружено, поэтому нужно сообщить JavaScript-коду, что все прошло успешно. Для этого вызывается указатель функции успеха:

```
UpdateHostOnSuccess();
```

Функция `ValidateName` в файле `validate.cpp` теперь должна выглядеть как код, показанный в листинге 6.1.

Для функции `IsCategoryIdInArray` никаких изменений не требуется. Внесите те же изменения, что и в функцию `ValidateName`, в функцию `ValidateCategory`, добавив

параметры — указатели на функции успеха и ошибки. Затем модифицируйте код, чтобы вызвать соответствующий указатель функции в зависимости от того, есть ли проблема с данными пользователя.

Листинг 6.1. Изменение функции ValidateName для использования указателей на функции (файл validate.cpp)

```
...

#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
void ValidateName(char* name, int maximum_length,
    OnSuccess UpdateHostOnSuccess, OnError UpdateHostOnError) {
  if (ValidateValueProvided(name) == 0) {
    UpdateHostOnError("A Product Name must be provided.");
  }
  else if (strlen(name) > maximum_length) {
    UpdateHostOnError("The Product Name is too long.");
  }
  else {
    UpdateHostOnSuccess();
  }
}
...
```

Теперь функция ничего не возвращает. Все выражения return были удалены

Были добавлены указатели на функции OnSuccess и OnError

Функция ValidateCategory

Измените тип возвращаемого значения функции `ValidateCategory`, чтобы теперь она возвращала `void`, а затем добавьте параметры — указатели на функции для случая успеха и для случая обнаружения проблемы с пользовательскими данными:

- `OnSuccess UpdateHostOnSuccess`;
- `OnError UpdateHostOnError`.

Удалите второй параметр из вызова функции `ValidateValueProvided` и замените строку `return 0` в этом операторе `if` следующим кодом:

```
UpdateHostOnError("A Product Category must be selected.");
```

Поскольку исходная функция JavaScript, `UpdateHostAboutError`, больше не вызывается, необходимо изменить вызовы, которые выполнялись для этой функции, чтобы вызвать указатель функции ошибки. Замените вызовы `UpdateHostAboutError` на `UpdateHostOnError` и удалите строку с оператором `return` в следующих местах:

- в операторе `if valid_category_ids == NULL`;
- в операторе `if IsCategoryIdInArray`.

Наконец, поскольку функция `ValidateCategory` теперь возвращает `void`, удалите строку `return 1` в конце функции и добавьте оператор `else` в конец блока `if`. Блок `else` сработает, если проблем с пользовательскими данными не было обнаружено. На данном этапе нужно сообщить коду JavaScript, что выбранная пользователем категория допустима, для чего вызывается указатель на функцию успеха:

```
UpdateHostOnSuccess();
```

Функция `ValidateCategory` в файле `validate.cpp` теперь должна выглядеть как код, показанный в листинге 6.2.

Листинг 6.2. Изменение функции ValidateCategory для использования указателей на функции (файл validate.cpp)

```
...

#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
void ValidateCategory(char* category_id, int* valid_category_ids,
    int array_length, OnSuccess UpdateHostOnSuccess,
    OnError UpdateHostOnError) {
  if (ValidateValueProvided(category_id) == 0) {
    UpdateHostOnError("A Product Category must be selected.");
  }
  else if ((valid_category_ids == NULL) || (array_length == 0)) {
    UpdateHostOnError("There are no Product Categories available.");
  }
  else if (IsCategoryIdInArray(category_id, valid_category_ids,
      array_length) == 0) {
    UpdateHostOnError("The selected Product Category is not valid.");
  }
  else {
    UpdateHostOnSuccess();
  }
}

...
```

Теперь функция ничего не возвращает. Все выражения return были удалены

Были добавлены указатели на функции OnSuccess и OnError

После того как код на C++ будет изменен для использования указателей на функции, можно сделать следующий шаг (рис. 6.5) и дать Emscripten указание скомпилировать код в модуль WebAssembly.

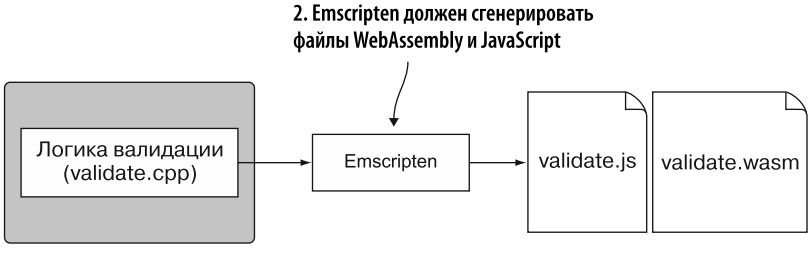


Рис. 6.5. Шаг 2 — командуем Emscripten сгенерировать файлы WebAssembly и JavaScript

6.1.3. Компиляция кода в модуль WebAssembly

Видя использование указателя на функцию C++, компилятор Emscripten ожидает, что функции с этими сигнатурами будут импортированы во время создания экземпляра модуля. После того как он будет создан, можно добавлять только экспортированные функции WebAssembly из другого модуля. Это значит, что в коде JavaScript нельзя будет задать указатель на функцию, которая еще не была импортирована.

Если вы не можете импортировать функции JavaScript после того, как экземпляр модуля будет создан, то как определить функцию JavaScript динамически? Оказывается, Emscripten предоставляет модулю функции, которые имеют необходимые сигнатуры во время создания экземпляра, а затем поддерживает вспомогательный массив в своем JavaScript-коде. Когда модуль вызывает указатель на функцию, Emscripten просматривает этот массив, чтобы узнать, предоставил ли ваш JavaScript-код функцию для вызова этой сигнатуры.

Для указателей на функции размер вспомогательного массива Emscripten должен быть явно установлен во время компиляции указанием параметра `RESERVED_FUNCTION_POINTERS`. Функции `ValidateName` и `ValidateCategory` ожидают два параметра указателей на функции, и мы изменим JavaScript для одновременного вызова обеих функций, поэтому вспомогательный массив должен будет содержать четыре элемента одновременно. В результате необходимо указать для этого параметра значение `4`.

Чтобы добавить или удалить указатели на функции из вспомогательного массива Emscripten, коду JavaScript потребуется доступ к вспомогательным функциям Emscripten — `addFunction` и `removeFunction`. Убедиться, что эти функции включены в сгенерированный файл JavaScript, можно, добавив их в массив командной строки `EXTRA_EXPORTED_RUNTIME_METHODS`.

Чтобы скомпилировать код в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой хранится файл `validate.cpp`, и выполните следующую команду:

```
emcc validate.cpp -s RESERVED_FUNCTION_POINTERS=4
➥ -s EXTRA_EXPORTED_RUNTIME_METHODS=['ccall','UTF8ToString',
➥'addFunction','removeFunction'] -o validate.js
```

Модуль WebAssembly и файл JavaScript в Emscripten сгенерированы. Следующий шаг (рис. 6.6) — копирование сгенерированных файлов в папку, в которой находятся файлы `editproduct.html` и `editproduct.js`, над которыми вы работали в главе 5. Затем нужно обновить файл `editproduct.js`, чтобы передать функции JavaScript в модуль.

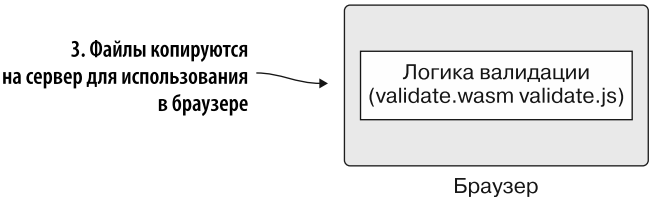


Рис. 6.6. Шаг 3 — копирование сгенерированных файлов в папку, в которой хранятся файлы HTML и JavaScript. Затем необходимо обновить JavaScript-код, чтобы передать функции JavaScript в модуль

6.1.4. Изменение JavaScript-кода веб-страницы

В папке Chapter 6\6.1.2 EmFunctionPointers\ создайте папку frontend, а затем скопируйте в нее следующие файлы:

- `validate.js` и `validate.wasm` из папки Chapter 6\6.1.2 EmFunctionPointers\source\;
- `editproduct.html` и `editproduct.js` из папки Chapter 5\5.1.1 EmJsLibrary\frontend\.

Откройте файл `editproduct.js` в своем любимом редакторе, чтобы изменить код для передачи указателей на функции в модуль.

Функция onClickSave

В коде на C++ вы изменили функции валидации модуля так, чтобы они больше не возвращали значения, а вместо этого вызывали предоставленные указатели на функции JavaScript с целью определить успех или ошибку, когда логика валидации готова сделать обратный вызов. Вы не знаете, когда будут вызваны указатели на функции, поэтому измените функции JavaScript `validateName` и `validateCategory`, чтобы они возвращали объект `Promise`.

Сейчас функция `onClickSave` использует оператор `if` для первого вызова функции `validateName`. Если с введенным пользователем названием проблем нет, то

оператор `if` вызывает функцию `validateCategory`. Поскольку обе функции будут изменены для возврата промиса, нужно будет модифицировать оператор `if` для работы с промисами.

Вы можете вызвать функцию `validateName`, дождаться ее успешного выполнения, а затем вызвать функцию `validateCategory`. Это сработает, но метод `Promise.all` будет вызывать обе функции валидации одновременно и упростит код по сравнению с выполнением одного вызова за раз.

Метод `Promise.all` передает массив промисов и возвращает один объект `Promise`. При успешном выполнении всех промисов вызывается метод `then`. Если какой-либо промис отклонен (произошла ошибка), то возвращается причина отклонения первого отклоненного промиса. Можно задействовать второй параметр метода `then` для получения причины отклонения, но вместо этого мы будем использовать оператор `catch` промиса, поскольку это наиболее распространенный подход, с помощью которого разработчики обрабатывают ошибки промисов.

Измените функцию `onClickSave` в файле `editproduct.js`, чтобы она соответствовала коду, показанному в листинге 6.3.

Листинг 6.3. Изменения в функции onClickSave для использования Promise.all (файл editproduct.js)

```
...

function onClickSave() {
  setErrorMessage("");

  const name = document.getElementById("name").value;
  const categoryId = getSelectedCategoryId();

  Promise.all([
    validateName(name),
    validateCategory(categoryId)
  ])
  .then(() => {

  })
  .catch((error) => {
    setErrorMessage(error);
  });
}
...
```

Вызывает обе функции валидации

Обе функции валидации вернули успех

Проблем при валидации не обнаружено. Данные могут быть сохранены

Если хотя бы одна из функций валидации возвращает ошибку, то вызывается этот блок

Отображает сообщение об ошибке

Прежде чем вы перейдете к изменению функций `validateName` и `validateCategory` для передачи функции JavaScript в модуль WebAssembly, мы разберемся, как передать функцию во вспомогательный массив Emscripten.

Вызов вспомогательной функции addFunction в Emscripten

Чтобы код JavaScript мог передать функцию модулю, он должен использовать функцию `addFunction` в Emscripten. Вызов `addFunction` добавит функцию JavaScript во вспомогательный массив, а затем вернет индекс, который необходимо передать функции `ccall`, как показано на рис. 6.7. (Дополнительную информацию о `ccall` можно найти в приложении Б.) Функция `addFunction` принимает два параметра:

- функцию JavaScript, которую нужно передать модулю;
- строку, представляющую сигнатуру функции.

Первый символ в строке сигнатуры функции — тип возвращаемого значения, а остальные символы — типы значений каждого параметра. Для типов значений доступны следующие символы:

- `v` — тип `void`;
- `i` — 32-битное целое число;
- `j` — 64-битное целое число;
- `f` — 32-битное число с плавающей запятой;
- `d` — 64-битное число с плавающей запятой.

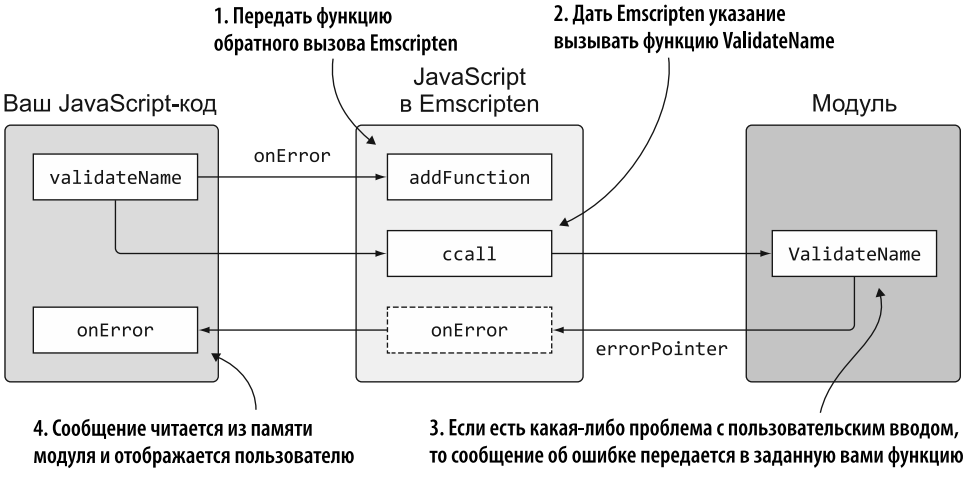


Рис. 6.7. Функция JavaScript передается во вспомогательный массив Emscripten, чтобы ее можно было вызвать из модуля

Когда ваш код заканчивает обработку указателя на функцию, нужно удалить его из вспомогательного массива Emscripten. Для этого передайте индекс, полученный от `addFunction`, в `removeFunction`.

Для каждой функции валидации модуля нужно будет передать два указателя на функции: один для обратного вызова при успехе, а другой для обратного вызова при ошибке валидации. Чтобы упростить задачу, создадим вспомогательную функцию JavaScript `createPointers`, которая поможет обеим функциям валидации JavaScript создавать указатели на функции.

Функция createPointers

Функция `createPointers` получит следующие параметры:

- `resolve` — метод `resolve` промиса, принадлежащего функции `validateName` или `validateCategory`;
- `reject` — метод `reject` промиса, принадлежащего функции `validateName` или `validateCategory`;
- `returnPointers` — объект, возвращаемый в вызывающую функцию и содержащий индекс каждой функции, которая была добавлена во вспомогательный массив Emscripten.

Воспользуемся *анонимными функциями* для обоих указателей на функции, которые будут добавлены во вспомогательный массив Emscripten.

СПРАВКА

Анонимные функции в JavaScript — функции, которые определены без указания имени. Больше информации доступно на странице MDN Web Docs: http://mng.bz/7zDV.

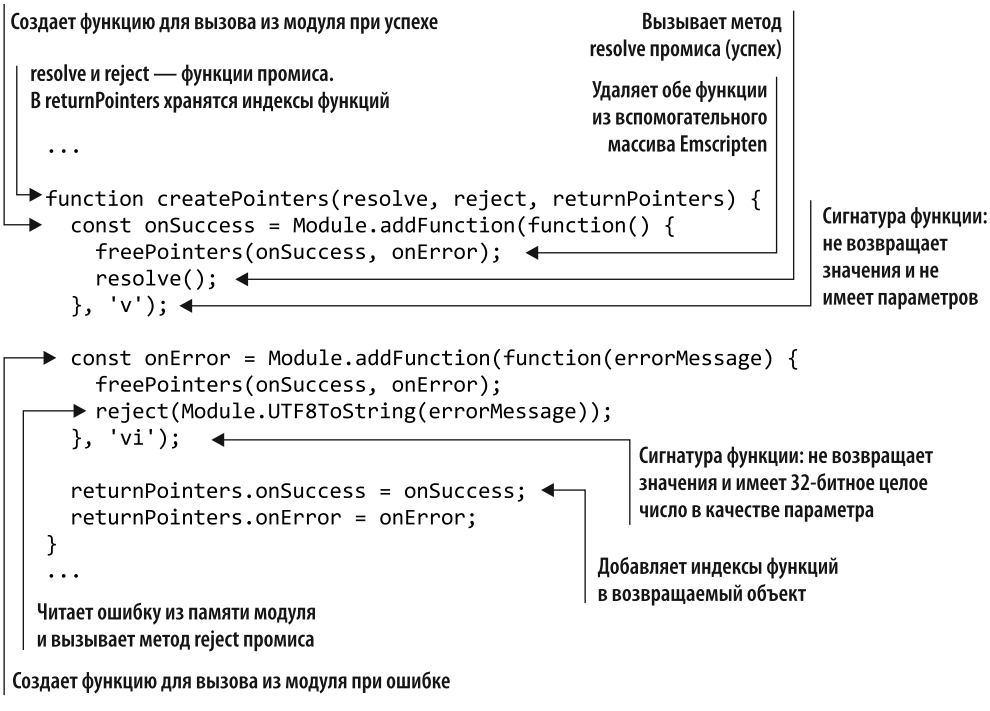
Указатель на функцию успеха, ожидаемый модулем, имеет тип возвращаемого значения `void` и не имеет параметров, поэтому значение, которое необходимо передать в качестве второго параметра функции `addFunction`, — `'v'`. При вызове данная функция сначала обратится к вспомогательной функции `freePointers`, а затем к методу `resolve`, который был передан в функцию `createPointers`.

Указатель на функцию ошибки, ожидаемый модулем, имеет возвращаемый тип `void` и параметр типа `const char*`. В WebAssembly указатели представлены 32-битными целыми числами. В данном случае строка сигнатуры функции, необходимая для второго параметра `addFunction`, — `'vi'`. При вызове эта функция сначала обратится к вспомогательной функции `freePointers`, прочитает сообщение об ошибке из памяти модуля, а затем вызовет метод `reject`, который был передан в функцию `createPointers`.

В конце функции `createPointers` индекс каждой функции, которую вы добавили во вспомогательный массив Emscripten, будет помещен в объект `returnPointers`.

После функции `onClickSave` в файле `editproduct.js` добавьте функцию `createPointers`, представленную в листинге 6.4.

Листинг 6.4. Новая функция createPointers в файле editproduct.js



Чтобы удалить указатели на функции из вспомогательного массива Emscripten после их обработки, создадим еще одну вспомогательную функцию `freePointers`.

Функция freePointers

После функции `createPointers` добавьте следующий фрагмент кода — функцию `freePointers` для удаления ваших функций из вспомогательного массива Emscripten:

```
function freePointers(onSuccess, onError){
  Module.removeFunction(onSuccess);
  Module.removeFunction(onError);
}
```

Удаляет функции из вспомогательного массива Emscripten

Теперь, после того как написаны функции, которые помогают добавлять функции во вспомогательный массив Emscripten и удалять их по завершении обработки, нужно изменить функции `validateName` и `validateCategory`. Модифицируем эти функции так, чтобы они возвращали объект `Promise` и с помощью новой функции `createPointers` передавали функции JavaScript в модуль.

Функция validateName

Изменим функцию `validateName`, чтобы она возвращала объект `Promise` — для этого будем использовать анонимную функцию внутри объекта `Promise`. В анонимной функции первое, что нужно сделать, — это вызвать функцию `createPointers`, чтобы создать функции `Success` и `Error`. Вдобавок вызов `createPointers` вернет индексы, которые необходимо передать модулю для указателей на функции успеха и ошибки. Эти индексы будут помещены в объект `pointers`, который передается в качестве третьего параметра функции `createPointers`.

Удалите код `const isValid =` перед `Module.ccall`, а затем измените функцию `Module.ccall` следующим образом:

- установите для второго параметра значение `null` с целью указать, что возвращаемое значение функции `ValidateName` недействительно;
- добавьте два дополнительных типа `'number'` в массив третьего параметра, поскольку функция модуля теперь принимает два новых параметра, которые являются указателями. Указатели в WebAssembly представлены 32-битными значениями, поэтому используется числовой тип;
- поскольку к функции модуля были добавлены два новых параметра, передайте индексы функций `Success` и `Error` четвертому параметру функции `ccall`. Индексы возвращаются в указателях объектов при вызове `createPointers`;
- удалите оператор `return` функции.

Функция `validateName` в файле `editproduct.js` теперь должна выглядеть как код, показанный в листинге 6.5.

Листинг 6.5. Измененная функция validateName в файле editproduct.js

```
...

function validateName(name) {
  return new Promise(function(resolve, reject) {

    const pointers = { onSuccess: null, onError: null };
    createPointers(resolve, reject, pointers);

    Module.ccall('ValidateName',
      null,
      ['string', 'number', 'number', 'number'],
      [name, MAXIMUM_NAME_LENGTH, pointers.onSuccess,
          pointers.onError]);

  });
}
...
```

Возвращает объект Promise

Создает указатели на функции для модуля

const isValid = удалено

Два новых типа 'number' были добавлены для двух новых параметров указателей

Индексы функций Success и Error добавлены в массив

Функция модуля теперь возвращает void

Внесите в функцию `validateCategory` те же изменения, что и в функцию `validateName`, — возврат объекта `Promise` и использование функции `createPointers` для создания указателей на функции, которые можно передать модулю.

Функция validateCategory

Как и для функции `validateName`, измените функцию `validateCategory`, чтобы она возвращала объект `Promise`. Вызовите функцию `createPointers`, чтобы создать функции `Success` и `Error`.

Удалите часть кода `const isValid =` перед функцией `Module.ccall`, а затем измените эту функцию следующим образом:

- измените второй параметр на `null`, поскольку функция модуля теперь возвращает `void`;
- добавьте два новых типа `'number'` в массив третьего параметра `ccall` для двух типов указателей;
- добавьте индексы функций `Success` и `Error` в массив четвертого параметра `ccall`;
- наконец, удалите оператор `return` из конца функции.

Ваша функция `validateCategory` должна выглядеть как код, показанный в листинге 6.6.

Листинг 6.6. Измененная функция validateCategory в файле editproduct.js

```
...

function validateCategory(categoryId) {
  return new Promise(function(resolve, reject) {

    const pointers = { onSuccess: null, onError: null };
    createPointers(resolve, reject, pointers);

    const arrayLength = VALID_CATEGORY_IDS.length;
    const bytesPerElement = Module.HEAP32.BYTES_PER_ELEMENT;
    const arrayPointer = Module._malloc((arrayLength * bytesPerElement));
    Module.HEAP32.set(VALID_CATEGORY_IDS,
        (arrayPointer / bytesPerElement));

    Module.ccall('ValidateCategory',
        null,
      ['string', 'number', 'number', 'number', 'number'],
      [categoryId, arrayPointer, arrayLength,
            pointers.onSuccess, pointers.onError]);

    Module._free(arrayPointer);

  });
}
```

Возвращает объект Promise

Создает указатели на функции для модуля

const isValid = удалено

Функция модуля теперь возвращает void

Два новых типа 'number' были добавлены для двух новых параметров указателей

Индексы функций Success и Error добавлены в массив

6.1.5. Просмотр результатов

Закончив изменять код JavaScript, вы можете открыть браузер и ввести `http://localhost:8080/editproduct.html` в адресную строку, чтобы увидеть получившуюся веб-страницу. Вы можете проверить валидацию, добавив более 50 символов в поле Name (Название) и нажав кнопку Save (Сохранить). На странице должна отобразиться ошибка (рис. 6.8).

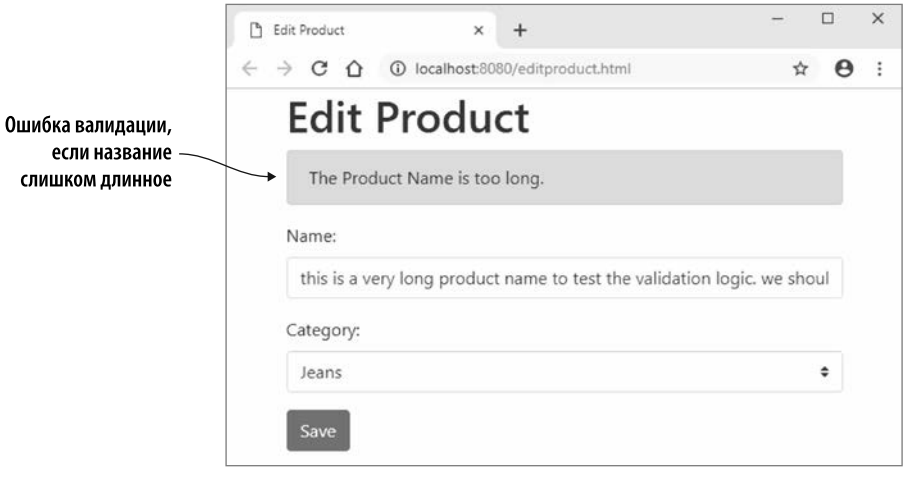


Рис. 6.8. Ошибка валидации на странице Edit Product (Изменить товар), если название слишком длинное

6.2. ИСПОЛЬЗОВАНИЕ C ИЛИ C++ ДЛЯ СОЗДАНИЯ МОДУЛЯ БЕЗ СВЯЗУЮЩЕГО ФАЙЛА EMSCRIPTEN

Предположим, вам нужно, чтобы Emscripten скомпилировал код на C++, но не добавил ни одну из стандартных функций библиотеки C или не сгенерировал связующий файл JavaScript. Связующий код Emscripten удобен и рекомендуется для использования в производственном коде, вдобавок он скрывает многие детали работы с модулями WebAssembly. Отсутствие связующего кода Emscripten позволяет работать напрямую с модулем WebAssembly.

Как вы можете видеть на рис. 6.9, процесс в этом разделе аналогичен процессу в разделе 6.1, за исключением того, что вы даете Emscripten указание сгенерировать только файл WebAssembly, без связующего файла JavaScript.

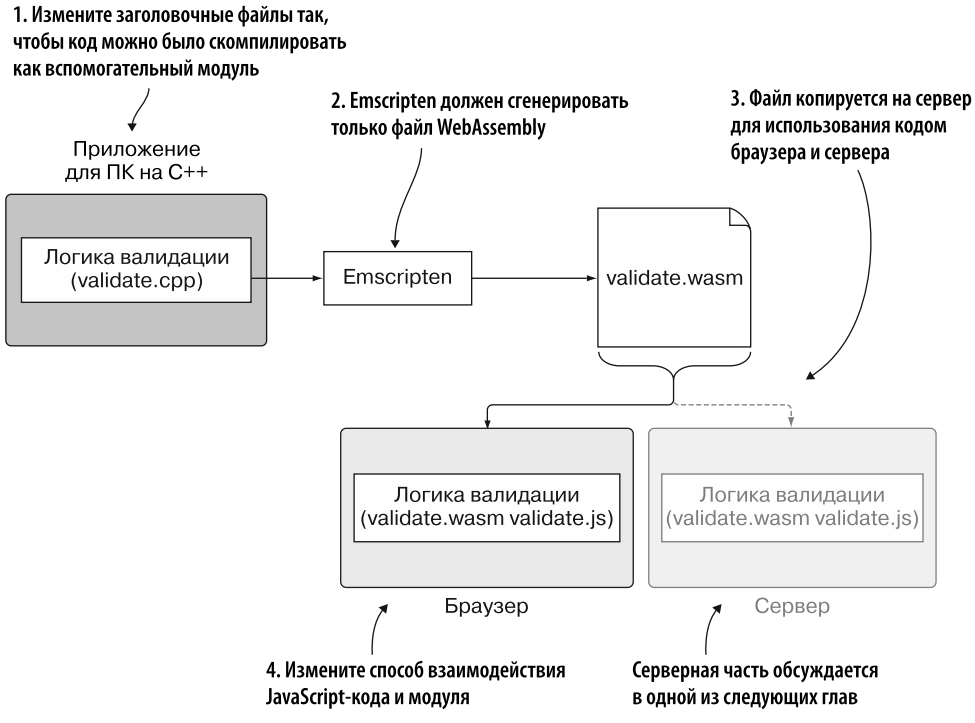


Рис. 6.9. Процесс преобразования логики на C++ в WebAssembly для использования кодом сайта и сервера, но без сгенерированного Emscripten JavaScript-кода. Серверная часть, Node.js, обсуждается в одной из следующих глав

6.2.1. Использование указателей на функции, передаваемых в модуль через JavaScript

При работе с указателями на функции в разделе 6.1 вы использовали связующий код Emscripten, который скрывал взаимодействие между модулем и JavaScript. Это выглядело так, что код JavaScript передает указатель на функцию в модуль.

Когда дело доходит до указателей на функции в WebAssembly, код на C или C++ пишется так, словно он вызывает указатели на функции напрямую. Однако при компиляции в модуль WebAssembly код фактически указывает индекс функции в разделе «Таблица» модуля и дает команду фреймворку WebAssembly вызывать функцию от его имени.

СПРАВКА

Раздел «Таблица» модуля — необязательный, но если присутствует, то содержит типизированный массив ссылок, таких как указатели на функции, которые не могут быть сохранены в памяти модуля в виде простых байтов. У модуля нет прямого доступа к элементам в разделе «Таблица». Вместо этого код запрашивает у фреймворка WebAssembly доступ к элементу на основе его индекса. Затем фреймворк обращается к памяти и работает с элементом от имени кода. Более подробно разделы модуля рассматриваются в главе 2.

Указатели на функции могут быть функциями внутри модуля или могут быть импортированы. В нашем случае, как показано на рис. 6.10, нужно указать функции для вызовов `OnSuccess` и `OnError`, чтобы можно было передавать сообщения обратно в JavaScript. Как и в случае со вспомогательным массивом Emscripten, ваш JavaScript-код должен поддерживать объект, содержащий ссылки на функции обратного вызова, которые необходимо вызывать, когда модуль вызывает функцию `OnSuccess` или `OnError`.

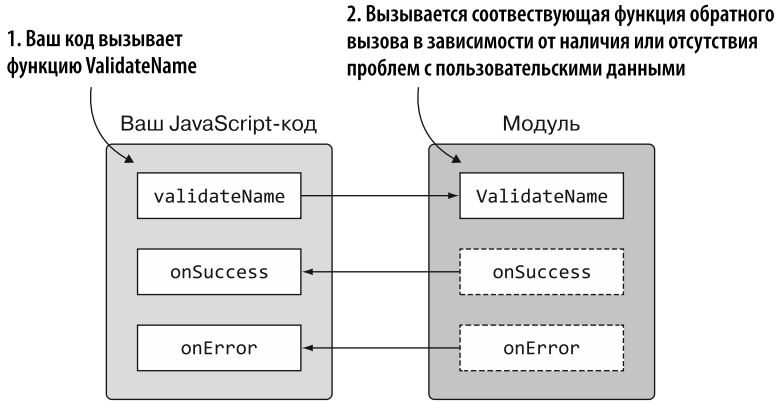


Рис. 6.10. Модуль, импортирующий функции onSuccess и onError в JavaScript в момент создания экземпляра. Когда функция модуля ValidateName вызывает другую функцию, она вызывает ее через JavaScript-код

6.2.2. Изменение кода на C++

Первый шаг процесса (рис. 6.11) — изменение кода на C++, добавленного в разделе 6.1, так, чтобы он использовал файлы `side_module_system_functions.h` и `.cpp`.

В папке Chapter 6\ создайте папку 6.2.2 SideModuleFunctionPointers\source\ для файлов в этом подразделе. Скопируйте в новую папку source следующие файлы:

- `validate.cpp` из папки 6.1.2 EmFunctionPointers\source\;
- `side_module_system_functions.h` и `.cpp` из папки Chapter 4\4.2 side_module\source\.

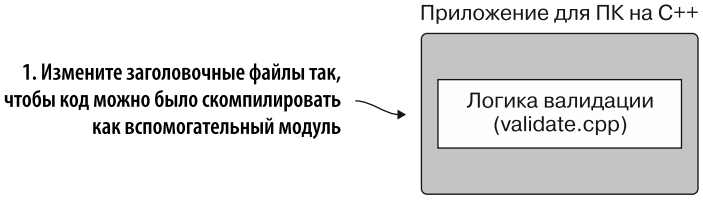


Рис. 6.11. Шаг 1 — изменение кода на C++ из раздела 6.1 так, чтобы модуль WebAssembly можно было сгенерировать без связующего кода Emscripten

Откройте файл `validate.cpp` в своем любимом редакторе.

Поскольку модуль WebAssembly будет создан как вспомогательный, Emscripten не будет включать стандартную библиотеку C, поэтому необходимо удалить строки подключения для заголовочных файлов `cstdlib` и `cstring`. Чтобы добавить в вашу версию стандартные функции библиотеки C для использования в коде, добавьте строку импорта для файла `side_module_system_functions.h` в блок `extern "C"`.

Первая часть вашего файла `validate.cpp` теперь должна выглядеть так:

```
#ifdef __EMSCRIPTEN__
  #include <emscripten.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif

#include "side_module_system_functions.h"
```

Важно: поместите этот заголовочный файл в блок extern "C"

Это все, что нужно изменить в файле `validate.cpp`. Остальной код оставим как есть.

6.2.3. Компиляция кода в модуль WebAssembly

После того как код на C++ будет изменен, можно сделать следующий шаг — дать Emscripten команду скомпилировать код в модуль WebAssembly, но без связующих файлов JavaScript, как показано на рис. 6.12.

Чтобы скомпилировать код на C++ в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой хранятся файлы C++, и выполните следующую команду:

```
emcc side_module_system_functions.cpp validate.cpp
➥ -s SIDE_MODULE=2 -O1 -o validate.wasm
```

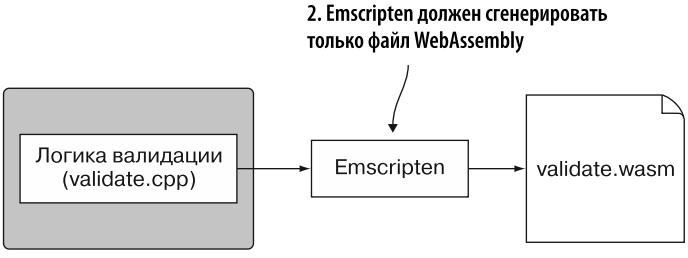


Рис. 6.12. Шаг 2 — командуем Emscripten сгенерировать только файл WebAssembly. В этом случае Emscripten не будет генерировать связующий файл JavaScript

6.2.4. Изменение JavaScript-кода, который будет взаимодействовать с модулем

На рис. 6.13 показан следующий шаг процесса — копирование сгенерированного файла Wasm в папку, в которой хранится файл HTML. Затем мы изменим способ взаимодействия JavaScript-кода с модулем, поскольку нам больше не нужен доступ к связующему коду Emscripten.

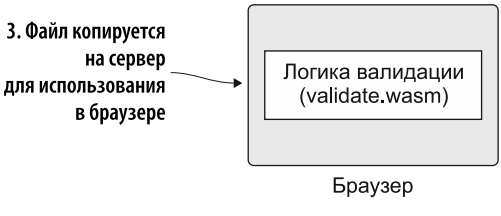


Рис. 6.13. Шаг 3 — копирование сгенерированного файла Wasm в папку, в которой хранится файл HTML, и изменение способа взаимодействия JavaScript-кода с модулем

В папке Chapter 6\6.2.2 SideModuleFunctionPointers\ создайте папку frontend\. Скопируйте в нее следующие файлы:

- `validate.wasm` из папки 6.2.2 SideModuleFunctionPointers\source\;
- `editproduct.html` и `editproduct.js` из папки Chapter 5\5.2.1 SideModuleCallingJS\frontend\.

Откройте файл `editproduct.js` в своем любимом редакторе, чтобы настроить код для работы с указателями на функции модуля WebAssembly.

Новые глобальные переменные

Нужно создать пару переменных, которые будут содержать позиции указателей на функции успеха и ошибки в разделе «Таблица» модуля. Поместите следующий

фрагмент кода между строками `const VALID_CATEGORY_IDS = [100, 101];` и `let moduleMemory = null;` в файле `editproduct.js`:

```
let validateOnSuccessNameIndex = -1;
let validateOnSuccessCategoryIndex = -1;
let validateOnErrorNameIndex = -1;
let validateOnErrorCategoryIndex = -1;
```

На время ожидания завершения обработки кода в модуле вам также понадобится некий способ отслеживать функции `resolve` и `reject` промисов из функций `validateName` и `validateCategory`. Для этого создадим объект для каждой функции, как показано в следующем фрагменте кода, который вы можете разместить после переменных, только что добавленных в файл `editproduct.js`:

```
let validateNameCallbacks = { resolve: null, reject: null };
let validateCategoryCallbacks = { resolve: null, reject: null };
```

Хотя ваш код на C++ выглядит так, словно вызывает указатель на функцию напрямую, в действительности это не так. Скрытым образом ссылки на указатели функций помещаются в раздел «Таблица» модуля. Код вызывает желаемую функцию по определенному индексу с помощью `call_indirect`, а WebAssembly вызывает функцию по этому индексу от имени кода. В JavaScript раздел «Таблица» представлен объектом `WebAssembly.Table`.

Нам также понадобится глобальная переменная для хранения экземпляра модуля `WebAssembly.Table`, которая будет передана модулю в целях хранения ссылок на указатели на функции. Поместите следующий код после строки `let moduleExports = null;` в файле `editproduct.js`:

```
let moduleTable = null;
```

Теперь, когда глобальные переменные созданы, перейдем к изменению функции `initializePage`, чтобы передать модулю ожидаемые объекты и функции.

Функция initializePage

Первое, что вам нужно сделать, — создать новый экземпляр объекта `WebAssembly.Table` для указателей на функции модуля. Этот объект ожидает от конструктора объект JavaScript.

Первое свойство объекта JavaScript называется `initial`, и оно указывает, каким должен быть начальный размер таблицы. Второе свойство — `element`, и единственное его значение, которое может быть предоставлено на данный момент, — это строка `funcref`. Существует третье необязательное свойство, называемое `maximum`. Оно указывает максимальный размер, до которого может увеличиваться таблица.

Первоначальное количество элементов, необходимых для таблицы, будет зависеть от компилятора Emscripten. Чтобы определить, какое значение использовать, вы можете включить флаг `-g` в командную строку при сборке модуля WebAssembly. Флаг даст Emscripten указание создать также файл текстового формата WebAssembly.

Если вы откроете сгенерированный файл текстового формата (`.wast`), то сможете выполнить поиск s-выражения `import` для объекта `table`, которое будет выглядеть примерно так:

```
(import "env" "table" (table $table 1 funcref))
```

В этом случае искомым значением будет `1`.

СПРАВКА

Спецификация WebAssembly была изменена, и теперь в качестве типа элемента таблицы служит слово funcref, а не anyfunc. Генерируя файл .wast, Emscripten использует новое имя, а WebAssembly Binary Toolkit теперь может принимать код текстового формата, который применяет любое имя. На момент написания этой книги инструменты разработчика в браузерах все еще задействовали слово anyfunc при валидации модуля. Firefox позволяет использовать любое слово при создании объекта WebAssembly.Table в JavaScript, однако на данный момент другие браузеры допускают только старое имя, поэтому JavaScript, применяемый в этой книге, будет по-прежнему задействовать anyfunc.

В функции `initializePage` после строки `moduleMemory` и непосредственно перед созданием `importObject` добавьте код, показанный в следующем фрагменте:

```
moduleTable = new WebAssembly.Table({initial: 1, element: "anyfunc"});
```

Затем добавьте некоторые свойства к `importObject`.

- После свойства `memory` добавьте свойство `__table_base` со значением `0` (ноль). Emscripten добавил этот импорт, поскольку в данном модуле будет раздел «Таблица», и ввиду того, что вспомогательные модули предназначены для динамического связывания, может быть несколько разделов таблицы, которые необходимо объединить. Здесь не выполняется динамическое связывание, поэтому можно просто передать ноль.
- После свойства `__table_base` нужно будет включить объект `table`, поскольку этот модуль использует указатели на функции, а ссылки на указатели на функции хранятся в разделе «Таблица» модуля.
- Функция `_UpdateHostAboutError` больше не нужна, поэтому ее можно удалить.
- Emscripten добавил строку импорта для функции `abort`, чтобы сообщить об имеющейся проблеме, препятствующей загрузке модуля. Вы предоставите

для него функцию, которая будет выбрасывать ошибку, указывающую на то, что функция `abort` была вызвана.

Внутри `then` функции `instantiateStreaming` нужно будет добавить вызовы к функции `addToTable` (мы сделаем это чуть позже) и передать анонимные функции для указателей на функции успеха и ошибки, которые будут вызываться функциями модуля `ValidateName` и `ValidateCategory`. Вторым параметром функции `addToTable` будет строка, представляющая сигнатуру добавляемой функции. Первый символ строки — это тип возвращаемого значения функции, а каждый дополнительный символ указывает типы параметров. Emscripten использует следующие символы:

- `v` — тип `void`;
- `i` — 32-разрядное целое число;
- `j` — 64-разрядное целое число;
- `f` — 32-разрядное число с плавающей запятой;
- `d` — 64-разрядное число с плавающей запятой.

Измените функцию `initializePage`, чтобы она выглядела как код, показанный в листинге 6.7.

Листинг 6.7. Изменения в функции initializePage (файл editproduct.js)

```
...

let moduleMemory = null;
let moduleExports = null;
let moduleTable = null;

function initializePage() {
  ...

  moduleMemory = new WebAssembly.Memory({initial: 256});
  moduleTable = new WebAssembly.Table({initial: 1,
      element: "anyfunc"});

  const importObject = {
    env: {
      __memory_base: 0,
      memory: moduleMemory,
      __table_base: 0,
      table: moduleTable,
      abort: function(i) { throw new Error('abort'); },
    }
  };

  WebAssembly.instantiateStreaming(fetch("validate.wasm"),
      importObject).then(result => {
    moduleExports = result.instance.exports;
```

anyfunc вместо funcref для старых браузеров

```
    validateOnSuccessNameIndex = addToTable(() => {
      onSuccessCallback(validateNameCallbacks);
    }, 'v');

    validateOnSuccessCategoryIndex = addToTable(() => {
      onSuccessCallback(validateCategoryCallbacks);
    }, 'v');

    validateOnErrorNameIndex = addToTable((errorMessagePointer) => {
      onErrorCallback(validateNameCallbacks, errorMessagePointer);
    }, 'vi');

    validateOnErrorCategoryIndex = addToTable((errorMessagePointer) => {
      onErrorCallback(validateCategoryCallbacks, errorMessagePointer);
    }, 'vi');
  });
}
...
```

В раздел «Таблица» были добавлены анонимные функции для указателей на функции успеха и ошибки

Теперь нужно создать функцию `addToTable`, которая добавит указанную функцию JavaScript в раздел «Таблица» модуля.

Функция addToTable

Функция `addToTable` сначала должна определить размер раздела «Таблица», поскольку это будет индекс, по которому нужно вставить функцию JavaScript. Метод `Grow` объекта `WebAssembly.Table` используется для увеличения размера раздела «Таблица» на желаемое количество элементов. Нам нужно добавить только одну функцию, поэтому определим, что таблица вырастет на `1` элемент.

Затем вызовем метод `set` объекта `WebAssembly.Table`, чтобы вставить функцию. Функции JavaScript нельзя передать объекту `WebAssembly.Table`, но можно экспортировать из другого модуля WebAssembly, поэтому передадим функцию JavaScript специальной вспомогательной функции (`convertJsFunctionToWasm`), которая преобразует функцию в функцию WebAssembly.

Добавьте следующий код после функции `initializePage` в файле `editproduct.js`:

```
  function addToTable(jsFunction, signature) {
    const index = moduleTable.length;
    moduleTable.grow(1);
    moduleTable.set(index,
        convertJsFunctionToWasm(jsFunction, signature));

    return index;
  }
```

Текущий размер определяет индекс новой функции

Увеличивает раздел «Таблица» на заданный размер, чтобы можно было добавить новую функцию

Возвращает индекс функции в разделе «Таблица»

Конвертирует функцию JavaScript в функцию Wasm и добавляет ее в раздел «Таблица»

Вместо того чтобы создавать функцию `convertJsFunctionToWasm`, скопируем функцию, используемую созданным Emscripten файлом JavaScript. Функция создает очень маленький модуль WebAssembly, который импортирует указанную вами функцию JavaScript. Модуль экспортирует ту же функцию, но теперь это функция в оболочке WebAssembly, которую можно вставить в объект `WebAssembly.Table`.

Откройте файл `validate.js` в папке Chapter 6\6.1.2 EmFunctionPointers\frontend\ и найдите функцию `convertJsFunctionToWasm`. Скопируйте ее и вставьте после функции `addFunctionToTable` в файле `editproduct.js`.

Следующая задача — создать вспомогательную функцию, которая используется модулем для индикации того, что валидация прошла успешно. Эта функция будет вызываться функциями модуля `ValidateName` и `ValidateCategory`, если проблем при валидации пользовательских данных не было обнаружено.

Функция onSuccessCallback

После функции `initializePage` в файле `editproduct.js` определите функцию `onSuccessCallback`, которая принимает в качестве параметра следующий объект в качестве параметра — `validateCallbacks`. Он будет ссылкой на глобальный объект `validateNameCallbacks` или `validateCategoryCallbacks`, в зависимости от того, вызывается ли эта функция для функции `validateName` или `validateCategory`. Внутри функции вызывается метод `resolve` callback-объекта, а затем из данного объекта удаляются функции.

Добавьте следующий фрагмент кода после функции `initializePage` в файле `editproduct.js`:

```
function onSuccessCallback(validateCallbacks) {
  validateCallbacks.resolve();
  validateCallbacks.resolve = null;
  validateCallbacks.reject = null;
}
```

Вызывает метод resolve промиса

Удаляет функции из объекта

Подобно только что созданной функции `onSuccessCallback`, вам нужно будет создать вспомогательную функцию, которая будет использоваться, когда модуль сообщает об ошибке при валидации пользовательских данных. Эта функция будет вызываться функциями модуля `ValidateName` и `ValidateCategory`.

Функция onErrorCallback

Следуя функции `onSuccessCallback` в файле `editproduct.js`, создадим функцию `onErrorCallback`, которая принимает два параметра:

- `validateCallbacks` — этот параметр представляет ссылку на глобальный объект `validateNameCallbacks` или `validateCategoryCallbacks`, в зависимости от того, вызывается ли эта функция для функции `validateName` или `validateCategory`;

- `errorMessagePointer` — указатель на место в памяти модуля, в котором находится сообщение об ошибке валидации.

Первое, что необходимо сделать функции, — прочитать строку из памяти модуля, вызвав вспомогательную функцию `getStringFromMemory`. Затем вызывается метод `reject` callback-объекта перед удалением функций из этого объекта.

Добавьте код, показанный ниже, после функции `onSuccessCallback` в файле `editproduct.js`:

```
function onErrorCallback(validateCallbacks, errorMessagePointer) {
  const errorMessage = getStringFromMemory(errorMessagePointer);

  validateCallbacks.reject(errorMessage);

  validateCallbacks.resolve = null;
  validateCallbacks.reject = null;
}
```

Вызывает метод reject промиса

Читает сообщение об ошибке из памяти модуля

Удаляет функции из объекта

Совсем скоро мы изменим функции `validateName` и `validateCategory` в JavaScript, чтобы они возвращали объект `Promise`, поскольку не знаем, когда модуль вызовет функции `Success` и `Error`. Поскольку функции будут возвращать объект `Promise`, функцию `onClickSave` необходимо будет изменить для работы с промисами.

Функция onClickSave

Измените функцию `onClickSave`, заменив оператор `if` кодом `Promise.all`, который вы видели в разделе 6.1. Измените код в функции `onClickSave` файла `editproduct.js`, чтобы он соответствовал коду, показанному в листинге 6.8.

Листинг 6.8. Измененная функция onClickSave (файл editproduct.js)

```
...

function onClickSave() {
  setErrorMessage("");

  const name = document.getElementById("name").value;
  const categoryId = getSelectedCategoryId();

  Promise.all([
    validateName(name),
    validateCategory(categoryId)
  ])
  .then(() => {

  })
```

Вызывает обе функции валидации

Обе функции валидации возвращают успех

При валидации проблем не обнаружено. Данные можно сохранить

```
  .catch((error) => {
    setErrorMessage(error);
  });
}
...
```

Если хотя бы одна функция валидации возвращает ошибку, …

то показываем ошибку валидации пользователю

Поскольку обе функции, `validateName` и `validateCategory`, должны иметь методы `resolve` и `reject` их объектов `Promise`, помещенные в глобальные переменные, создадим вспомогательную функцию `createPointers`, которую они обе могут использовать.

Функция createPointers

После функции `onClickSave` добавьте функцию `createPointers`, которая принимает следующие параметры:

- `isForName` — флаг, который указывает, вызывается ли функция `validateName` или `validateCategory`;
- `resolve` — метод `resolve` промиса вызывающей функции;
- `reject` — метод `reject` промиса вызывающей функции;
- `returnPointers` — объект, который вы будете использовать для возврата индекса функций `_OnSuccess` и `_OnError`, которые должна вызывать функция модуля.

На основе значения `isForName` вы поместите методы `resolve` и `reject` в соответствующий callback-объект.

Функция модуля должна знать, какой индекс в разделе «Таблица» модуля необходимо вызвать для указателей функций `_OnSuccess` и `_OnError`. Поместим правильный индекс в объект `returnPointers`.

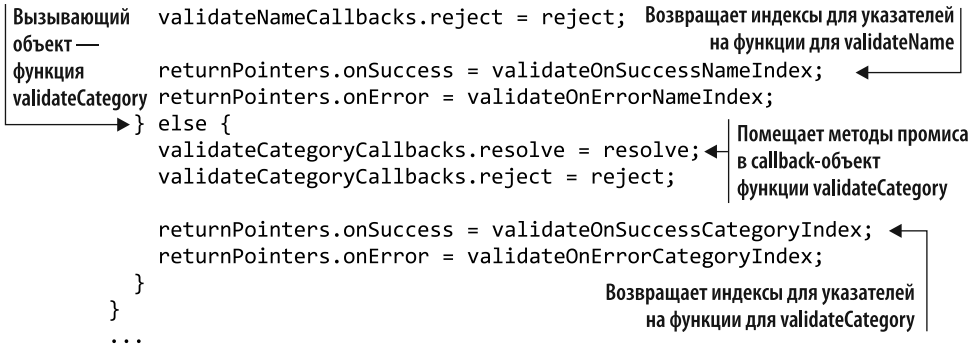
Добавьте код, показанный в листинге 6.9, после функции `onClickSave` в файле `editproduct.js`.

Листинг 6.9. Функция createPointers (файл editproduct.js)

Помещает методы промиса в callback-объект функции validateName

Вызывающий объект — функция validateName

```
...

function createPointers(isForName, resolve, reject, returnPointers) {
  if (isForName) {
    validateNameCallbacks.resolve = resolve;
```



Теперь нужно изменить функции `validateName` и `validateCategory`, чтобы они возвращали объект `Promise`, и с помощью новой функции `createPointers` дать возможность функции модуля вызвать соответствующий указатель на функцию.

Функция validateName

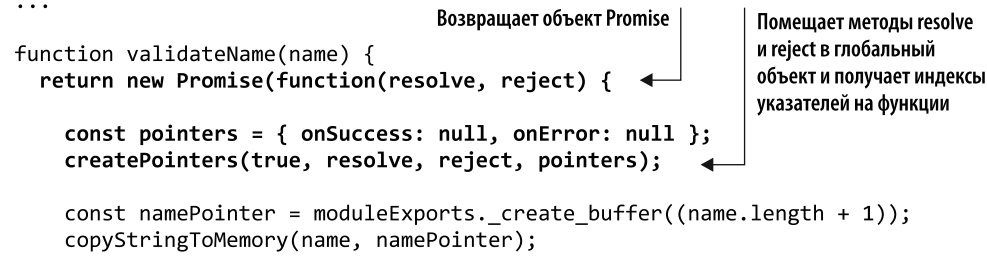
Измените функцию `validateName`, которая теперь будет возвращать объект `Promise`. Содержимое промиса будет обернуто в анонимную функцию.

Потребуется добавить вызов функции `createPointers`, чтобы методы `resolve` и `reject` промиса были помещены в глобальный объект `validateNameCallbacks`. Вызов объекта `createPointers` также вернет правильные индексы для передачи функции модуля `_ValidateName`, чтобы она вызвала указатель на функцию `_OnSuccessName` или `_OnErrorName`.

Функция модуля `_ValidateName` больше не возвращает значение, поэтому нужно удалить часть кода `const isValid =`, а также оператор `return` в конце функции. Вызов функции `_ValidateName` необходимо изменить еще и затем, чтобы получить два индекса указателей на функции.

Измените функцию `validateName` в файле `editproduct.js`, чтобы она соответствовала коду, показанному в листинге 6.10.

Листинг 6.10. Изменения в функции validateName (файл editproduct.js)



```
    moduleExports._ValidateName(namePointer, MAXIMUM_NAME_LENGTH,
        pointers.onSuccess, pointers.onError);

    moduleExports._free_buffer(namePointer);
  });
}
...
```

Передает индексы указателей на функции в _OnSuccessName и _OnErrorName

Те же изменения, что и для функции `validateName`, нужно внести в функцию `validateCategory`.

Функция validateCategory

Единственная разница в изменениях здесь заключается в том, что вы указываете `false` в качестве первого параметра функции `createPointers`, чтобы она знала, что вызывается функция `validateCategory`, а не `validateName`.

Измените функцию `validateCategory` в файле `editproduct.js`, чтобы она соответствовала коду, показанному в листинге 6.11.

Листинг 6.11. Изменения в функции validateCategory (файл editproduct.js)

```
...

function validateCategory(categoryId) {
  return new Promise(function(resolve, reject) {

    const pointers = { onSuccess: null, onError: null };
    createPointers(false, resolve, reject, pointers);

    const categoryIdPointer =
    ➥ moduleExports._create_buffer((categoryId.length + 1));
    copyStringToMemory(categoryId, categoryIdPointer);

    const arrayLength = VALID_CATEGORY_IDS.length;
    const bytesPerElement = Int32Array.BYTES_PER_ELEMENT;
    const arrayPointer = moduleExports._create_buffer((arrayLength *
    ➥ bytesPerElement));

    const bytesForArray = new Int32Array(moduleMemory.buffer);
    bytesForArray.set(VALID_CATEGORY_IDS,
        (arrayPointer / bytesPerElement));

    moduleExports._ValidateCategory(categoryIdPointer, arrayPointer,
         arrayLength, pointers.onSuccess, pointers.onError);

    moduleExports._free_buffer(arrayPointer);
    moduleExports._free_buffer(categoryIdPointer);
  });
}
```

Возвращает объект Promise

Помещает методы resolve и reject в глобальный объект и получает индексы указателей на функции

Передает индексы указателей на функции в _OnSuccessCategory и _OnErrorCategory

6.2.5. Просмотр результатов

Завершив изменение кода, вы можете открыть браузер и ввести `http://localhost:8080/editproduct.html` в адресную строку, чтобы просмотреть получившуюся веб-страницу. Протестируйте валидацию, изменив выбор в раскрывающемся списке Category (Категория) так, чтобы ничего не было выбрано, а затем нажав кнопку Save (Сохранить). Валидация должна отображать ошибку на веб-странице, как показано на рис. 6.14.

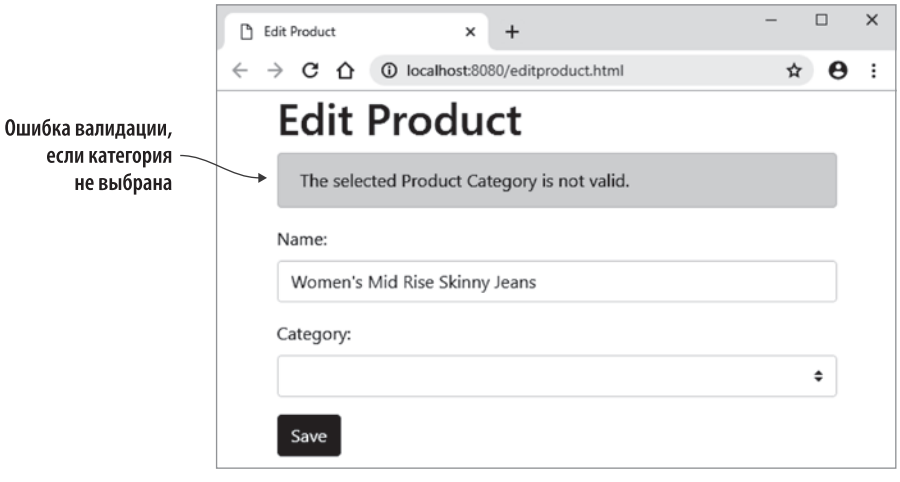


Рис. 6.14. Ошибка валидации при выборе категории на странице Edit Product (Изменить товар)

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже приведены некоторые возможные варианты использования того, что вы узнали в этой главе.

- С помощью указателей на функции вы можете создавать функции JavaScript, которые возвращают промис, что позволяет модулю работать так же, как и другие методы JavaScript, например `fetch`. Возвращая объект `Promise`, ваша функция даже может быть связана с другими промисами.
- Если заданный указатель функции имеет ту же сигнатуру, что ожидает модуль WebAssembly, то его можно вызвать. Например, это позволяет коду модуля использовать одну и ту же сигнатуру для `onSuccess` в каждой функции. Код

JavaScript может указывать две или более функции, которые соответствуют этой сигнатуре, и в зависимости от того, какой код JavaScript вызывает функцию, заставить модуль вызвать ту функцию `onSuccess`, которая соответствует текущему действию.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какие две функции используются для добавления и удаления указателей функций из вспомогательного массива Emscripten?
2. Какую инструкцию WebAssembly применяет для вызова функции, определенной в разделе «Таблица»?

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Можно определить сигнатуру указателя функции непосредственно в параметре функции в C или C++.
- Можно определить сигнатуру с помощью ключевого слова `typedef`, а затем использовать определенное имя сигнатуры в параметрах функции.
- В действительности указатели на функции не вызываются напрямую кодом WebAssembly. Вместо этого ссылки на функции хранятся в разделе «Таблица» модуля и код указывает фреймворку WebAssembly вызвать желаемую функцию по заданному индексу.

Часть III

Продвинутые темы

Теперь, изучив основы создания модулей WebAssembly и работы с ними, мы можем перейти к части книги, в которой рассматриваются способы уменьшения объема загружаемых данных и улучшения возможностей повторного использования, а также преимущества параллельной обработки или даже применения модулей WebAssembly вне браузера.

Глава 7 знакомит вас с основами динамического связывания, при котором два или более модуля WebAssembly могут быть связаны в среде выполнения в целях использования функций друг друга.

Глава 8 дополняет сведения, полученные из главы 7, — в ней вы познакомитесь с тем, как создавать несколько экземпляров одного и того же модуля WebAssembly и динамически связывать каждый экземпляр с другим модулем WebAssembly по мере надобности.

В главе 9 вы узнаете, как в случае необходимости предварительно загружать модули WebAssembly с помощью веб-воркеров. Вы также изучите, как выполнять параллельную обработку с использованием потоков pthread в модуле WebAssembly.

Глава 10 демонстрирует, что WebAssembly не ограничивается браузером. В этой главе вы узнаете, как применять модули WebAssembly в Node.js.

7 Динамическое связывание: основы

В этой главе

- ✓ Как работает динамическое связывание модулей WebAssembly.
- ✓ Когда стоит использовать динамическое связывание и когда не стоит.
- ✓ Как создавать модули WebAssembly в качестве основных и вспомогательных модулей.
- ✓ Какие есть подходы к применению динамического связывания и как использовать каждый из них.

Говоря о *динамическом связывании* модулей WebAssembly, мы имеем в виду процесс объединения двух или более модулей в среде выполнения, когда неразрешенные символы из одного модуля (например, функции) разрешаются в символы, существующие в другом модуле. Исходное количество модулей WebAssembly не меняется, но теперь они связаны друг с другом и имеют доступ к функциям друг друга, как показано на рис. 7.1.

Динамическое связывание для модулей WebAssembly можно реализовать несколькими способами, поэтому данная тема достаточно обширна. Вы узнаете, как создать сайт, использующий динамическое связывание, в главе 8, но сначала мы рассмотрим все доступные варианты.

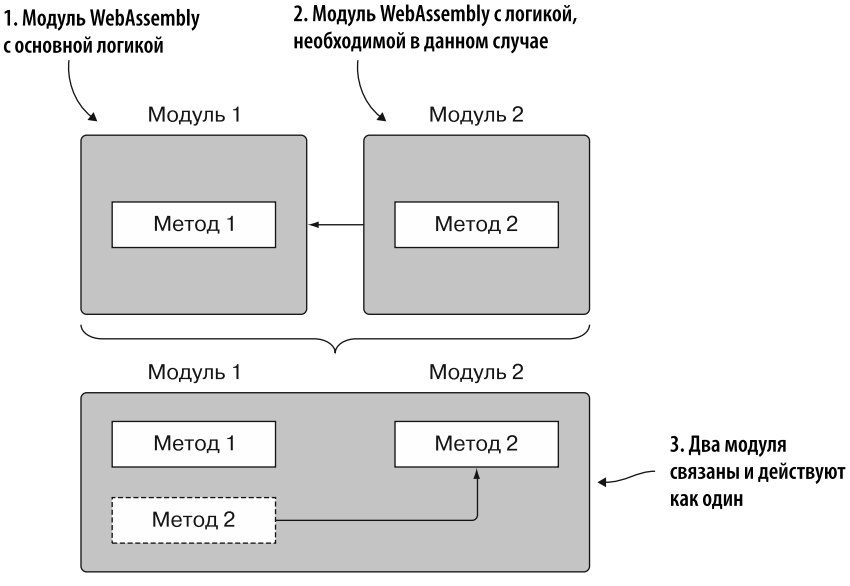


Рис. 7.1. В среде выполнения логика одного модуля (в данном случае модуля 2) связана с другим модулем (модуль 1), что позволяет им взаимодействовать и выполняться как единому целому

7.1. ДИНАМИЧЕСКОЕ СВЯЗЫВАНИЕ: ЗА И ПРОТИВ

Зачем использовать динамическое связывание нескольких модулей вместо одного модуля WebAssembly, как мы делали до сих пор? Динамическое связывание может вам пригодиться по нескольким причинам:

- Ускорение разработки. Вместо того чтобы компилировать один большой модуль, достаточно скомпилировать только измененные модули.
- Ядро вашего приложения может быть выделено, чтобы им было легче поделиться. Вместо двух или трех больших модулей WebAssembly с одинаковой логикой в каждом вы можете задействовать базовый модуль с несколькими меньшими модулями, связанными с ним. Примером такого подхода могут быть игровые движки — движок можно загрузить отдельно от игры, и несколько игр могут использовать один и тот же движок.
- Чем меньше объект, тем быстрее он загружается, поэтому загрузка только необходимого кода потребует меньше времени. Если веб-странице требуется дополнительная логика, то можно загрузить меньший модуль с логикой, специфичной для данной области.

- Если часть логики никогда не используется, то не будет загружена, поскольку логика загружается только по мере необходимости. В результате мы сэкономим время на загрузку и обработку ненужного кода.
- Браузер кэширует модуль аналогично кэшированию изображений или файлов JavaScript. Повторно загружаются лишь измененные модули, что ускоряет последующие просмотры страниц, поскольку необходимо загружать заново только часть логики.

Хотя динамическое связывание имеет ряд преимуществ, его нельзя применять всегда и везде, поэтому лучше проверить, годится ли такой подход в вашем случае.

Динамическое связывание может повлиять на производительность. Согласно документации Emscripten, снижение производительности может составлять от 5 до 10 % или выше, в зависимости от того, как структурирован ваш код. Ниже представлены несколько примеров, в которых вы можете заметить влияние на производительность:

- при разработке настройка сборки проекта становится более сложной, поскольку теперь нужно создать два и более модуля WebAssembly вместо одного;
- вместо загрузки одного модуля WebAssembly вам понадобится как минимум два модуля, а это значит, что и количество сетевых запросов увеличится;
- модули необходимо связать, поэтому во время создания экземпляров понадобится больше времени на обработку;
- поставщики браузеров работают над улучшением производительности для различных типов вызовов, но, согласно Emscripten, вызовы функций между связанными модулями могут выполняться медленнее, чем вызовы внутри модуля. Если количество вызовов между связанными модулями велико, то могут возникнуть проблемы с производительностью.

Теперь, изучив плюсы и минусы динамического связывания, посмотрим, как его можно реализовать с помощью модулей WebAssembly.

7.2. ВАРИАНТЫ РЕАЛИЗАЦИИ ДИНАМИЧЕСКОГО СВЯЗЫВАНИЯ

Доступно три варианта динамического связывания с использованием Emscripten:

- ваш код на C или C++ вручную связывается с модулем с помощью функции `dlopen`;

- вы сообщаете Emscripten, что существуют связываемые модули WebAssembly, добавив их в массив `dynamicLibraries` в файле JavaScript, созданном Emscripten. При создании экземпляров модулей WebAssembly Emscripten автоматически загружает и связывает модули, указанные в этом массиве;
- в вашем коде JavaScript экспорт одного модуля вручную передается другому как импорт с помощью JavaScript API в WebAssembly.

СПРАВКА

Краткий обзор JavaScript API в WebAssembly можно найти в главе 3. На странице MDN Web Docs http://mng.bz/vln1 также есть хороший обзор.

Прежде чем вы узнаете, как использовать каждый из методов динамического связывания, рассмотрим, в чем состоит разница между вспомогательными и основными модулями.

7.2.1. Вспомогательные и основные модули

В предыдущих главах мы создавали модули WebAssembly как вспомогательные, чтобы не создавать файл JavaScript в Emscripten. Это позволяет вручную загружать и создавать экземпляры модулей WebAssembly с помощью JavaScript API в WebAssembly. Создание вспомогательного модуля в целях ручного использования API — полезный побочный эффект, помогающий понять, как все работает «за кулисами», однако в действительности вспомогательные модули предназначены для динамического связывания.

При использовании вспомогательных модулей Emscripten опускает функции стандартной библиотеки C и файл JavaScript, поскольку вспомогательные модули будут связаны с основным в среде выполнения (рис. 7.2). Основной модуль будет содержать созданный Emscripten файл JavaScript и стандартные функции библиотеки C; при подключении вспомогательный модуль получает доступ к функциям основного.

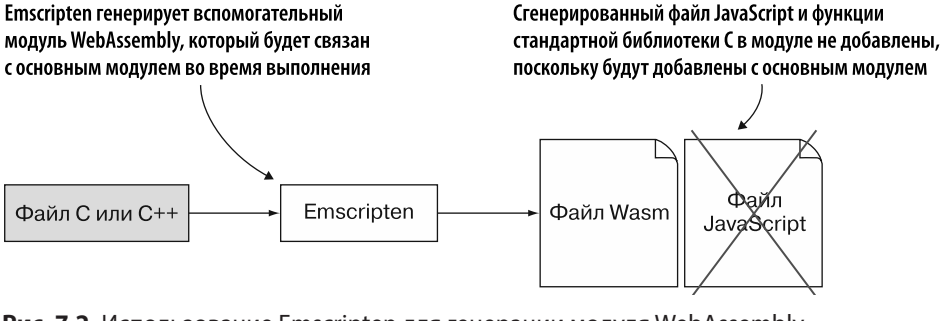


Рис. 7.2. Использование Emscripten для генерации модуля WebAssembly как вспомогательного. В этом случае функции стандартной библиотеки C не включаются в модуль и JavaScript-файл не генерируется

Вспомогательные модули создаются путем добавления параметра `SIDE_MODULE` в командную строку, чтобы дать Emscripten указание не генерировать файл JavaScript и не включать какие-либо функции стандартной библиотеки C в модуль.

Основные модули создаются так же, как и вспомогательные, но с помощью параметра командной строки `MAIN_MODULE`. Он дает компилятору Emscripten указание добавлять системные библиотеки и логику, необходимые для динамического связывания. Как показано на рис. 7.3, основной модуль будет сгенерирован вместе с файлом JavaScript, созданным Emscripten, и будет включать в себя функции стандартной библиотеки C.

ПРИМЕЧАНИЕ

При динамическом связывании следует помнить о том, что, хотя несколько вспомогательных модулей могут быть связаны с основным, основной модуль может быть только один. Однако это не означает, что в основном модуле обязательно должна располагаться функция main(), — ее можно разместить в любом из модулей, включая вспомогательный.

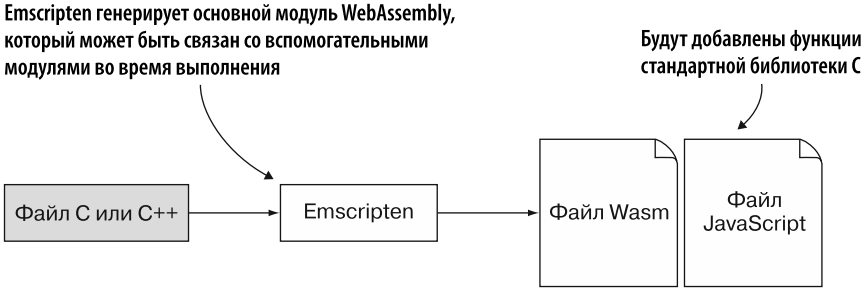


Рис. 7.3. Использование Emscripten для генерации модуля WebAssembly как основного модуля. Функции стандартной библиотеки C добавлены в модуль, и файл JavaScript также в этом случае генерируется в Emscripten

Первый тип динамического связывания, который мы рассмотрим, — подход `dlopen`.

7.2.2. Динамическое связывание: dlopen

Предположим, начальник попросил вас создать модуль WebAssembly. Одна из функций этого модуля — определить простые числа, которые существуют в заданном диапазоне. Оглядываясь назад, вы можете вспомнить, что уже создавали эту логику в главе 3 как обычный модуль WebAssembly (`calculate_primes.c`).

Не стоит просто копировать и вставлять логику в новый модуль WebAssembly, поскольку нет ничего хорошего в том, чтобы поддерживать два идентичных набора кода. При обнаружении проблемы придется изменять одну и ту же логику в двух местах, что может привести к пропуску одной из частей, если разработчик не знает о ее существовании, или к неправильным изменениям в какой-то из них.

Вместо того чтобы дублировать код, нам нужно изменить существующий код `calculate_primes`, чтобы его можно было использовать как обычный модуль WebAssembly, а также вызывать его из нового модуля WebAssembly. Как показано на рис. 7.4, необходимо выполнить следующие действия.

1. Изменить файл `calculate_primes.c`, созданный в главе 3, так, чтобы его мог вызывать и основной модуль. Переименовать файл `calculate_primes.cpp`.
2. Использовать Emscripten для создания файла WebAssembly из файла `calculate_primes.cpp` в качестве вспомогательного модуля.
3. Создать логику (`main.cpp`), которая будет связываться с вспомогательным модулем, используя вызов функции `dlopen`.
4. Использовать Emscripten для создания файла WebAssembly из файла `main.cpp` в качестве основного модуля и создания файла шаблона HTML.

В этом сценарии мы будем вызвать функцию `dlopen` из кода на C++ для связи со вспомогательным модулем `calculate_primes`. Однако для того, чтобы открыть вспомогательный модуль, `dlopen` требуется нахождение файла WebAssembly в файловой системе Emscripten.

Но вот в чем загвоздка с файловой системой — модуль WebAssembly работает в виртуальной машине и не имеет доступа к файловой системе устройства. Чтобы обойти это, Emscripten предоставляет модулю WebAssembly один из нескольких различных типов файловой системы в зависимости от места запуска модуля (в браузере или в Node.js, например) и от того, насколько постоянным должно быть хранилище. По умолчанию файловая система Emscripten находится в памяти и любые записанные в нее данные будут потеряны при обновлении веб-страницы.

Доступ к файловой системе Emscripten осуществляется через объект `FS` в созданном Emscripten файле JavaScript, но этот объект добавляется лишь в том случае, если код модуля WebAssembly обращается к файлам. (Чтобы узнать больше о файловой системе Emscripten, обратитесь к https://emscripten.org/docs/api_reference/Filesystem-API.html.) В данной главе вы узнаете только, как использовать функцию `emscripten_async_wget`, которая позволит загрузить модуль WebAssembly в файловую систему Emscripten, чтобы его можно было открыть с помощью функции `dlopen`.

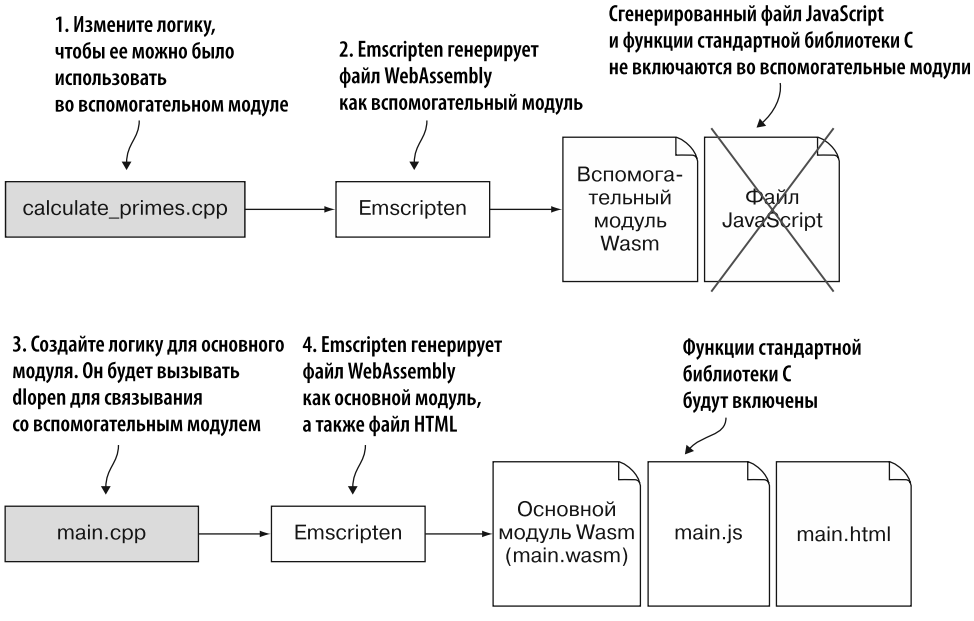


Рис. 7.4. Порядок изменения файла calculate_primes.cpp, для того чтобы его можно было скомпилировать во вспомогательный модуль WebAssembly, и шаги по созданию основного модуля WebAssembly, который будет связан со вспомогательным модулем через вызов функции dlopen

При использовании подхода `dlopen` для динамического связывания ваш модуль сможет вызывать функцию `main` в модуле `calculate_primes`, даже когда она есть и в нем самом. Это может быть полезно, если модуль поставляется сторонними системами и содержит логику инициализации. Возможность вызвать функцию `main` в другом модуле доступна, поскольку `dlopen` возвращает дескриптор вспомогательного модуля, а затем вы получаете ссылку на функцию, которую нужно вызвать на основе данного дескриптора.

СОВЕТ

Это одно из преимуществ использования подхода dlopen для динамического связывания по сравнению с подходом dynamicLibraries, описанным ниже. Когда дело доходит до применения второго подхода, вызов функции в другом модуле при наличии функции с таким же именем в вашем модуле не сработает. В итоге вы сможете вызвать только функцию в своем модуле, что может привести к рекурсивному вызову.

Первый шаг процесса реализации динамического связывания (рис. 7.5) — изменение файла `calculate_primes.cpp`, чтобы его можно было скомпилировать во вспомогательный модуль.

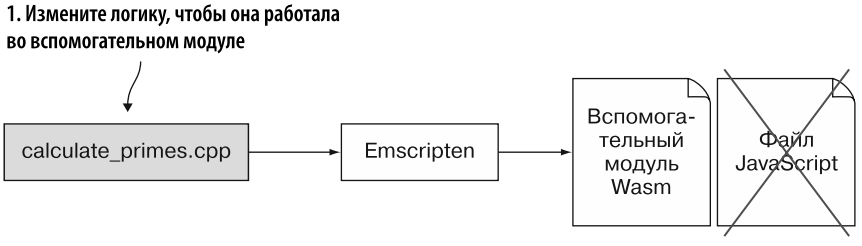


Рис. 7.5. Шаг 1 реализации динамического связывания с помощью dlopen — изменение файла calculate_primes.cpp так, чтобы его можно было скомпилировать во вспомогательный модуль

Изменения в файле calculate_primes.cpp

В папке WebAssembly\ создайте папку Chapter 7\7.2.2 dlopen\source\ для файлов, которые будут использоваться в этом подразделе. Скопируйте файл `calculate_primes.c` из папки Chapter 3\3.5 js_plumbing\source\ в созданную папку source\ и измените расширение файла на `.cpp`. Откройте файл `calculate_primes.cpp` в своем любимом редакторе.

Замените заголовочный файл `stdlib.h` на `cstdlib` и заголовочный файл `stdio.h` на `cstdio`; затем добавьте открывающий блок `extern "C"` между заголовочным файлом `emscripten.h` и перед функцией `IsPrime`. Теперь начало файла `calculate_primes.cpp` должно соответствовать коду, показанному ниже:

```
#include <cstdlib>      ← Замена заголовочного файла stdlib.h
#include <cstdio>       ← Замена заголовочного файла stdio.h
#include <emscripten.h>

#ifdef __cplusplus      ← Добавление открывающего блока extern "C"
extern "C" {
#endif
```

В файле `calculate_primes.cpp` после функции `IsPrime` и перед функцией `main` создайте функцию `FindPrimes`, которая возвращает `void` и принимает два целочисленных параметра (`start` и `end`) — начало и конец диапазона поиска простых чисел.

Удалите строки объявления переменных начала и конца из функции `main`, а затем переместите оставшийся код, за исключением строки `return 0`, из функции `main` в `FindPrimes`.

Добавьте объявление `EMSCRIPTEN_KEEPALIVE` перед функцией `FindPrimes`, чтобы она автоматически добавлялась в список экспортируемых функций при компиляции. Это упрощает работу при использовании Emscripten для создания модуля WebAssembly, поскольку вам не нужно явно указывать функцию в командной строке.

Измените функцию `main` так, чтобы вызвать новую функцию `FindPrimes` и передать исходный диапазон от `3` до `100000`. Наконец, после функции `main` добавьте закрывающую скобку для блока `extern "C"`.

Новая функция `FindPrimes`, измененная функция `main` и закрывающая скобка для блока `extern "C"` должны теперь соответствовать коду, показанному в листинге 7.1.

Листинг 7.1. Новая функция FindPrimes и измененная функция main

```
...
EMSCRIPTEN_KEEPALIVE
void FindPrimes(int start, int end) {
  printf("Prime numbers between %d and %d:\n", start, end);

  for (int i = start; i <= end; i += 2) {
    if (IsPrime(i)) {
      printf("%d ", i);
    }
  }
  printf("\n");
}

int main() {
  FindPrimes(3, 100000);

  return 0;
}

#ifdef __cplusplus
}
#endif
```

Новую функцию теперь можно экспортировать и вызывать в других модулях

Начальный диапазон простых чисел

Добавление закрывающей скобки для блока extern "C"

После того как код будет изменен так, чтобы другие модули могли его вызывать, можно сделать шаг 2 (рис. 7.6) и скомпилировать код во вспомогательный модуль WebAssembly.

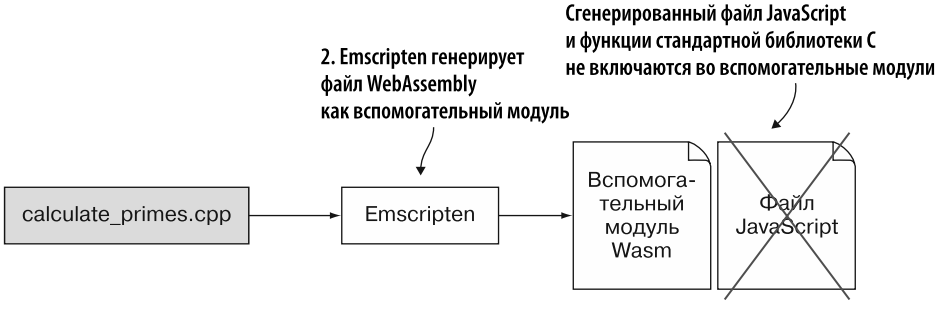


Рис. 7.6. Шаг 2 — использование Emscripten для генерации файла WebAssembly как вспомогательного модуля

Использование Emscripten для генерации файла WebAssembly как вспомогательного модуля из calculate_primes.cpp

В предыдущих главах, создавая вспомогательные модули WebAssembly, вы заменяли функции стандартной библиотеки C неким заменяющим кодом, добавленным в главе 4. Это было сделано для того, чтобы вспомогательный модуль продолжал работать, даже если функции стандартной библиотеки C недоступны. В этом случае заменяющий код не понадобится, поскольку вспомогательный модуль будет связан с основным в среде выполнения, а основной модуль будет иметь функции стандартной библиотеки C.

Чтобы скомпилировать измененный файл `calculate_primes.cpp` как вспомогательный модуль WebAssembly, откройте командную строку, перейдите в папку Chapter 7\7.2.2 dlopen\source\ и выполните следующую команду:

```
emcc calculate_primes.cpp -s SIDE_MODULE=2 -O1
➥ -o calculate_primes.wasm
```

После того как вспомогательный модуль будет создан, можно сделать следующий шаг — создать основной модуль (рис. 7.7).

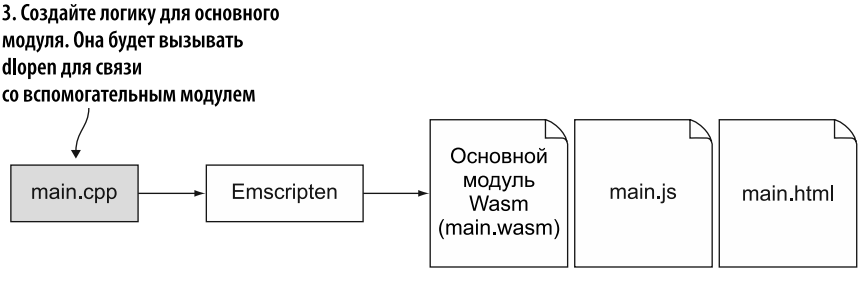


Рис. 7.7. Шаг 3 при реализации динамического связывания с помощью dlopen — создание логики, которая будет использовать dlopen для связи со вспомогательным модулем

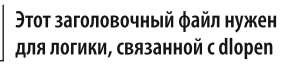
Создание логики для связи со вспомогательным модулем

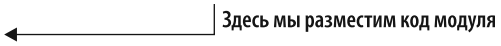
В папке Chapter 7\7.2.2 dlopen\source\ создайте файл `main.cpp`, а затем откройте его в своем любимом редакторе. Первое, что нужно добавить в файл `main.cpp`, — заголовочные файлы. В этом случае добавьте к `cstdlib` и `emscripten.h` заголовочный файл `dlfcn.h`, поскольку в нем есть объявления, относящиеся к динамическому связыванию при использовании `dlopen`. После этого вам нужно добавить блок `extern "C"`.

Код в файле `main.cpp` теперь должен выглядеть так, как показано в листинге 7.2.

Листинг 7.2. Файл main.cpp с включением заголовочных файлов и блоком extern "C"

```
#include <cstdlib>

#ifdef __EMSCRIPTEN__
  #include <dlfcn.h>
  #include <emscripten.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif

#ifdef __cplusplus
}
#endif
```





В коде, который вы собираетесь написать, функция `dlopen` будет использоваться для получения дескриптора вспомогательного модуля WebAssembly. Затем потребуется с помощью функции `dlsym` получить указатель на нужную функцию в данном модуле. Чтобы упростить код при вызове функции `dlsym`, определите сигнатуру функции для `FindPrimes`, которая будет вызываться во вспомогательном модуле.

Функция `FindPrimes` возвращает `void` и имеет два целочисленных параметра. Сигнатура указателя функции для функции `FindPrimes` показана в следующем фрагменте кода, который необходимо включить в файл `main.cpp` внутри блока `extern "C"`:

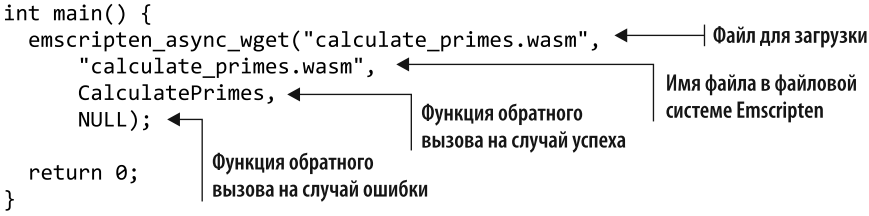
```
typedef void(*FindPrimes)(int,int);
```

Теперь добавьте в файл функцию `main`, чтобы компилятор Emscripten добавил ее в начальную часть модуля WebAssembly. Это приведет к автоматическому запуску функции `main` после того, как будет создан экземпляр модуля.

В функции `main` добавьте вызов функции `emscripten_async_wget`, чтобы загрузить вспомогательный модуль в файловую систему Emscripten. Это асинхронный вызов — он передаст управление в указанную вами функцию обратного вызова после завершения загрузки. Ниже перечислены параметры функции `emscripten_async_wget` в нужном порядке.

1. Файл для загрузки: `"calculate_primes.wasm"`.
2. Имя, которое будет дано файлу при добавлении в файловую систему Emscripten. В данном случае ему будет присвоено имя загружаемого файла.
3. Функция обратного вызова на случай успешной загрузки — `CalculatePrimes`.
4. В этом случае четвертый параметр функции будет равен `NULL`, поскольку функция обратного вызова не указана. Если нужно, то можете указать функцию обратного вызова на случай ошибки при загрузке файла.

Следуя сигнатуре указателя функции `FindPrimes` в файле `main.cpp`, внутри блока `extern "C"` добавьте следующий код:



Последнее, что нужно добавить в файл `main.cpp`, — функцию, которая будет содержать логику для открытия вспомогательного модуля, получать ссылку на функцию `FindPrimes`, а затем вызывать ее.

Когда функция `emscripten_async_wget` завершит загрузку модуля WebAssembly `calculate_primes`, она вызовет указанную функцию `CalculatePrimes` и передаст параметр — имя загруженного файла. Чтобы открыть вспомогательный модуль, нам понадобится функция `dlopen` со следующими значениями параметров:

- имя открываемого файла из параметра имени файла, полученного функцией `CalculatePrimes`;
- целое число, указывающее *режим*: `RTLD_NOW`.

ОПРЕДЕЛЕНИЕ

Когда исполняемый файл переносится в адресное пространство процесса, он может иметь ссылки на символы, которые останутся неизвестными, пока не будет загружен файл. Эти ссылки необходимо переместить, чтобы получить доступ к символам. Значение режима используется, чтобы сообщить dlopen, когда должно произойти перемещение. Значение RTLD_NOW указывает dlopen, что перемещение нужно выполнить при загрузке файла. Дополнительную информацию о dlopen и флагах режима можно найти в базовых спецификациях Open Group по адресу http://mng.bz/4eDQ.

Вызов функции `dlopen` вернет дескриптор файла, как показано ниже:

```
void* handle = dlopen(file_name, RTLD_NOW);
```

Получив дескриптор вспомогательного модуля, вы вызовете функцию `dlsym` со следующими значениями параметров, чтобы получить ссылку на функцию, которую нужно вызвать:

- дескриптор вспомогательного модуля;
- имя функции, на которую нужна ссылка: `"FindPrimes"`.

Функция `dlsym` вернет указатель на запрошенную функцию:

```
FindPrimes find_primes = (FindPrimes)dlsym(handle, "FindPrimes");
```

Получив указатель на функцию, вы можете вызывать его так же, как и обычную функцию. Как только работа со связанным модулем будет завершена, вы можете освободить его, передав дескриптор файла в функцию `dlclose`.

Учитывая все вышесказанное, функция `CalculatePrimes` должна выглядеть как код, показанный в листинге 7.3. Добавьте этот код в файл `main.cpp` между сигнатурой указателя функции `FindPrimes` и функцией `main`.

Листинг 7.3. Функция CalculatePrimes, которая вызывает функцию во вспомогательном модуле

```
...
void CalculatePrimes(const char* file_name) {
  void* handle = dlopen(file_name, RTLD_NOW);
  if (handle == NULL) { return; }

  FindPrimes find_primes =
      (FindPrimes)dlsym(handle, "FindPrimes");
  if (find_primes == NULL) { return; }

  find_primes(3, 100000);

  dlclose(handle);
}
...
```

Открывает вспомогательный модуль

Получает ссылку на функцию FindPrimes

Вызывает функцию из вспомогательного модуля

Закрывает вспомогательный модуль

После того как код для основного модуля будет добавлен, можно сделать последний шаг (рис. 7.8) — выполнить компиляцию в модуль WebAssembly. Вы также можете дать Emscripten указание сгенерировать файл шаблона HTML.

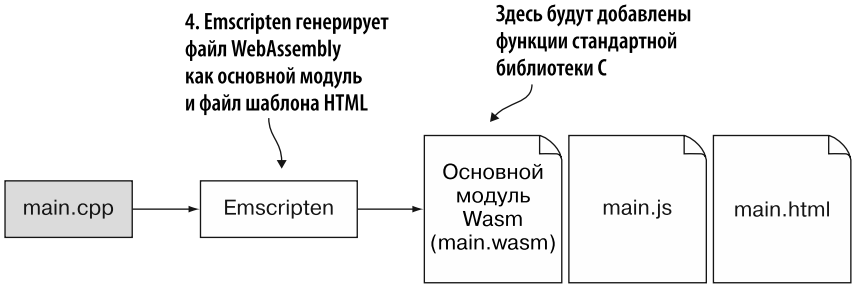


Рис. 7.8. Шаг 4 при реализации динамического связывания с помощью dlopen — использование Emscripten для генерации модуля WebAssembly как основного из файла main.cpp. В этом случае Emscripten должен еще сгенерировать файл шаблона HTML

Использование Emscripten для генерации файла WebAssembly как основного модуля из main.cpp

Вместо того чтобы создавать HTML-страницу для просмотра результатов, будем использовать HTML-шаблон Emscripten, указав выходной файл с расширением `.html`. Чтобы скомпилировать файл `main.cpp` в основной модуль, нужно добавить флаг `-s MAIN_MODULE=1`. К сожалению, если бы вы просматривали сгенерированную HTML-страницу, используя только следующую команду, то увидели бы ошибку, показанную на рис. 7.9:

```
emcc main.cpp -s MAIN_MODULE=1 -o main.html
```

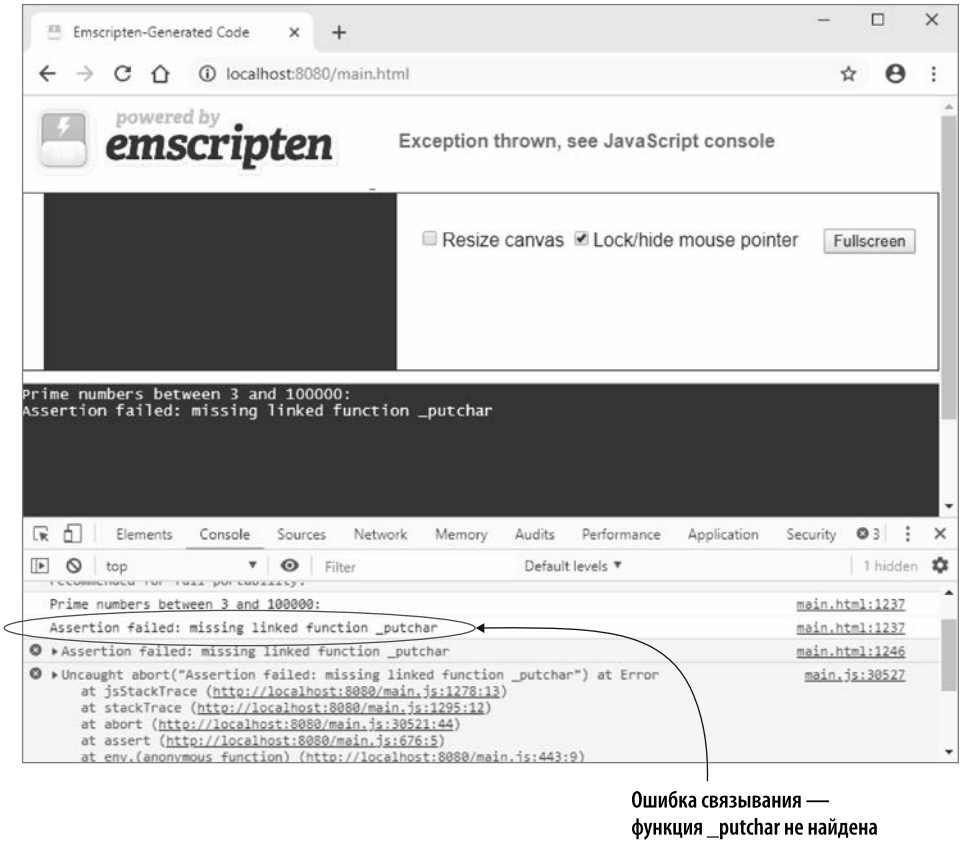


Рис. 7.9. При просмотре веб-страницы выбрасывается ошибка связывания — функция _putchar не найдена

Вы можете видеть, что модуль WebAssembly был загружен и `dlopen` без проблем связалась со вспомогательным модулем, поскольку текст `"Prime numbers between`

`3 and 100000"` («Простые числа от 3 до 100 000») выводится функцией `FindPrimes` во вспомогательном модуле. Если бы возникла проблема с динамическим связыванием, то код не достиг бы этой точки. Ни одно из простых чисел не было выведено на экран, это значит, проблема находится в функции `FindPrimes` вспомогательного модуля, но после вызова `printf` для указания диапазона.

Оказывается, проблема заключается в том, что в файле `calculate_primes.cpp` функция `printf` используется для вывода только одного символа. В данном случае символ переноса строки (`\n`) в конце функции `FindPrimes` приводит к ошибке. Функция `printf` внутри себя использует функцию `putchar`, которая не добавлена по умолчанию.

Есть три варианта исправления этой ошибки.

- Добавьте функцию `_putchar` в массив `EXPORTED_FUNCTIONS` как часть командной строки при создании модуля WebAssembly. При тестировании этого возможного исправления добавление данной функции приведет к исчезновению ошибки, но, к сожалению, на веб-странице ничего не будет отображаться. Если вы воспользуетесь этим подходом, то потребуется также добавить в массив функцию `_main` модуля.
- Вы можете изменить вызов `printf` в файле `calculate_primes.cpp` так, чтобы он выводил как минимум два символа, чтобы предотвратить использование функции `putchar` внутренним образом для вызова `printf`. Проблема этого подхода заключается в том, что если `printf` с использованием одного символа применяется где-либо еще, то ошибка повторится снова. Следовательно, данное исправление не рекомендуется.
- Вы можете добавить флаг `-s EXPORT_ALL=1`, чтобы Emscripten включал все символы при создании модуля WebAssembly и файла JavaScript. Это сработает, но использовать данный подход рекомендуется, только если нет других обходных путей — поскольку в этом случае размер сгенерированного файла JavaScript увеличивается вдвое лишь для того, чтобы экспортировать одну функцию.

К сожалению, все три подхода выглядят лишь как временные «костыли» для решения проблемы. Первый подход кажется наиболее подходящим, поэтому для исправления ошибки будем использовать массив командной строки `EXPORTED_FUNCTIONS`, чтобы модуль экспортировал функции `_putchar` и `_main`.

Чтобы скомпилировать файл `main.cpp` в основной модуль WebAssembly, откройте командную строку, перейдите в папку `Chapter 7\7.2.2 dlopen\source\` и выполните следующую команду:

```
emcc main.cpp -s MAIN_MODULE=1
➥ -s EXPORTED_FUNCTIONS=['_putchar','_main'] -o main.html
```

После того как будут созданы модули WebAssembly, можно просмотреть результаты.

Просмотр результатов

Откройте браузер и введите `http://localhost:8080/main.html` в адресную строку, чтобы увидеть созданную веб-страницу. Как показано на рис. 7.10, она должна отображать список простых чисел как в текстовом поле, так и в окне консоли инструментов разработчика браузера. Простые числа определяются вспомогательным модулем, который вызывает функцию `printf` — часть основного модуля.

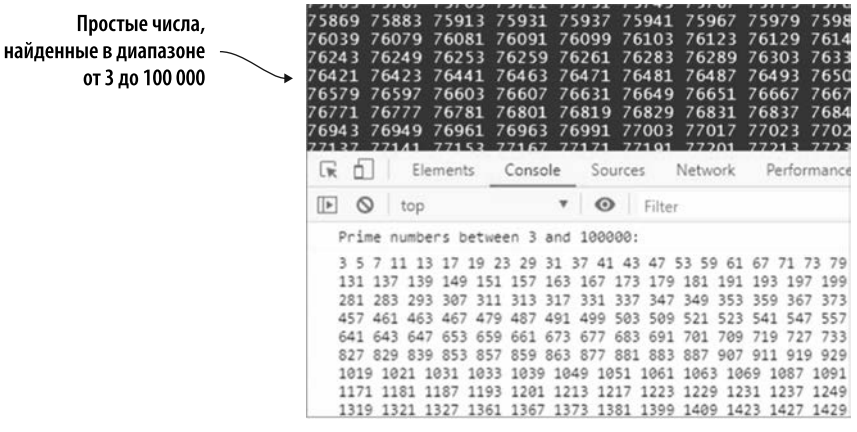


Рис. 7.10. Простые числа определяются вспомогательным модулем, который вызывает функцию printf — часть основного модуля

Теперь вы узнали, как выполнять динамическое связывание с помощью `dlopen`, поэтому можно перейти к подходу, в котором используется `dynamicLibraries`.

7.2.3. Динамическое связывание: dynamicLibraries

Представьте, что ваши коллеги и начальник смогли увидеть новые модули WebAssembly в действии. Их весьма впечатлили ваши действия с `dlopen`, но начальник почитал о динамическом связывании, пока вы создавали модули, и обнаружил, что это связывание также можно реализовать с помощью массива `dynamicLibraries` от Emscripten. Вашему начальнику стало интересно узнать, каков подход `dynamicLibraries` в сравнении с `dlopen`, поэтому вас попросили оставить вспомогательный модуль `calculate_primes` как есть, но создать основной модуль, который связывается с ним с помощью `dynamicLibraries`.

Как показано на рис. 7.11, шаги для этого сценария будут следующими.

1. Создайте логику (`main.cpp`), которая будет взаимодействовать со вспомогательным модулем.

2. Создайте файл JavaScript, который будет включен в созданный Emscripten JavaScript-файл, чтобы сообщить Emscripten о вспомогательном модуле, с которым его нужно связать.
3. Используйте Emscripten для создания файла WebAssembly из `main.cpp` в качестве основного модуля, а также для создания файла шаблона HTML.

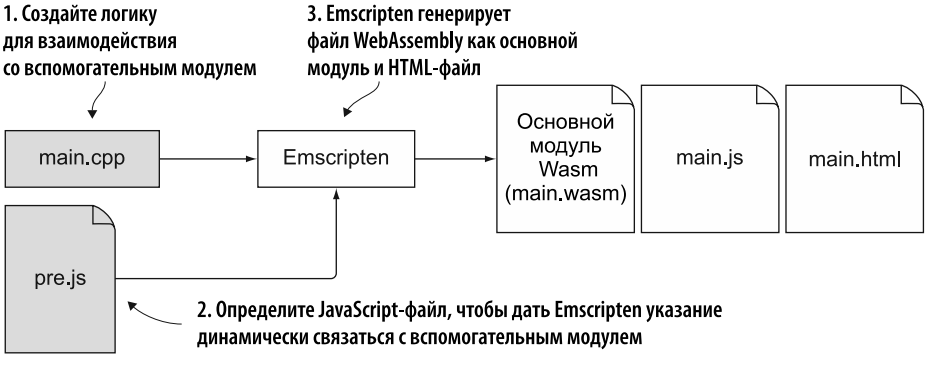


Рис. 7.11. Шаги по созданию основного модуля WebAssembly, который будет указывать массиву dynamicLibraries в Emscripten, с каким вспомогательным модулем его нужно динамически связать

Создание логики для взаимодействия со вспомогательным модулем

Для этого сценария первым шагом процесса (рис. 7.12) является создание файла `main.cpp`, который будет содержать логику, взаимодействующую со вспомогательным модулем. В папке Chapter 7\ создайте папку 7.2.3 dynamicLibraries\source\. В ней:

- скопируйте файл `calculate_primes.wasm` из папки 7.2.2 dlopen\source\;
- создайте файл `main.cpp`, а затем откройте его в своем любимом редакторе.

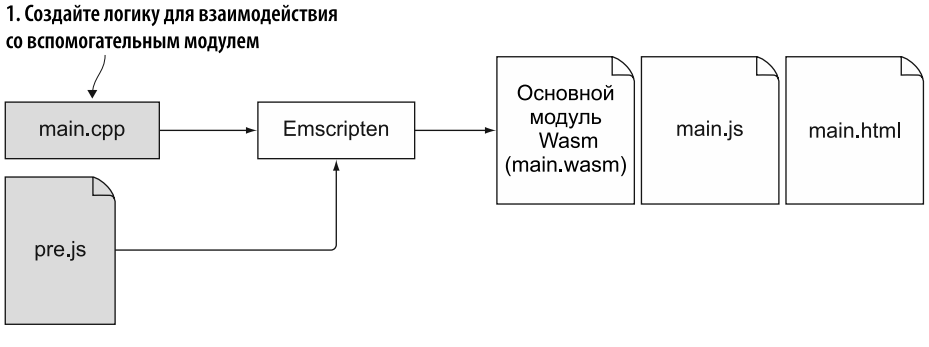


Рис. 7.12. Шаг 1 для реализации динамического связывания с помощью dynamicLibraries — создание файла main.cpp

Добавьте заголовочные файлы для стандартной библиотеки C и Emscripten. Затем добавьте блок `extern "C"`. Код в файле `main.cpp` теперь должен выглядеть так, как показано в листинге 7.4.

Листинг 7.4. Файл main.cpp с заголовочным файлом и блоком extern "C"

```
#include <cstdlib>

#ifdef __EMSCRIPTEN__
  #include <emscripten.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif
                    Здесь мы разместим код модуля

#ifdef __cplusplus
}
#endif
```

Сейчас мы напишем функцию `main`, которая будет вызывать функцию `FindPrimes` во вспомогательном модуле `calculate_primes`. Поскольку функция `FindPrimes` — часть другого модуля, необходимо добавить ее сигнатуру с префиксом `extern`, чтобы компилятор знал, что функция будет доступна при запуске кода.

Добавьте следующую сигнатуру функции в блок `extern "C"` в файле `main.cpp`:

```
extern void FindPrimes(int start, int end);
```

Последнее, что нужно сделать в файле `main.cpp`, — добавить функцию `main`, чтобы код запускался автоматически при создании экземпляра модуля WebAssembly. В функции `main` вы просто вызываете функцию `FindPrimes`, передавая ей диапазон чисел от `3` до `99`.

Добавьте следующий фрагмент кода в файл `main.cpp` внутри блока `extern "C"`, но после сигнатуры функции `FindPrimes`:

```
int main() {
  FindPrimes(3, 99);

  return 0;
}
```

Теперь код на C++ готов к превращению в модуль WebAssembly. Прежде чем использовать Emscripten для этого, необходимо создать JavaScript-код, который будет указывать Emscripten, что нужно выполнить связывание с вашим вспомогательным модулем (рис. 7.13).

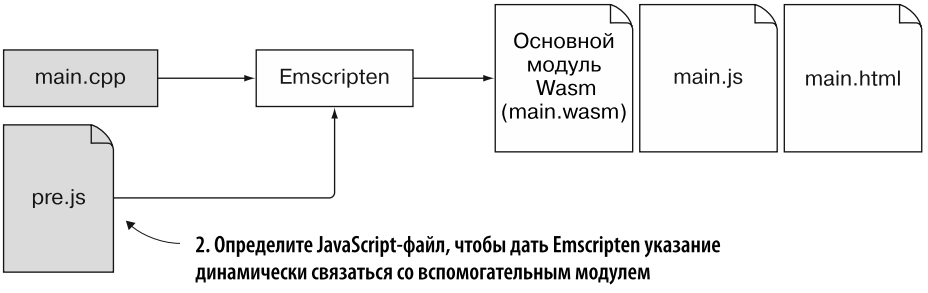


Рис. 7.13. Шаг 2 при реализации динамического связывания с помощью dynamicLibraries — создание JavaScript-кода, который даст Emscripten указание выполнить связывание со вспомогательным модулем

Создание JavaScript-файла, для того чтобы указать Emscripten, с каким вспомогательным модулем нужно его связать

Поскольку ваш начальник просто хочет понять, в чем разница между подходами `dlopen` и `dynamicLibraries`, мы создадим модуль WebAssembly и дадим Emscripten указание сгенерировать HTML-шаблон для его запуска вместо создания собственной HTML-страницы.

Чтобы связать вспомогательный модуль с основным, применив подход `dynamicLibraries`, вам нужно написать JavaScript-код для указания вспомогательного модуля, с которым будет связываться Emscripten. Для этого имена файлов вспомогательных модулей указываются в массиве `dynamicLibraries` в Emscripten до того, как Emscripten создаст экземпляр модуля.

При использовании HTML-шаблона Emscripten можно добавить JavaScript в начале создаваемого Emscripten JavaScript-файла, указав его в командной строке с помощью флага `--pre-js` при создании модуля WebAssembly. Если бы мы создавали собственную веб-страницу, то можно было бы указать настройки для файла JavaScript, созданного Emscripten, такие как массив `dynamicLibraries`, в объекте `Module` перед тегом `script` HTML-страницы. Когда Emscripten загружает файл JavaScript, тот создает собственный объект `Module`; но если объект `Module` уже существует, то значения из этого объекта будут скопированы в новый объект `Module`.

ИНФОБОКС

Можно установить ряд настроек, управляющих выполнением JavaScript-кода, созданного Emscripten. На веб-странице https://emscripten.org/docs/api_reference/module.html перечислены некоторые из них.

Если вы используете шаблон HTML, созданный Emscripten, то в нем определяется объект `Module` для реагирования на определенные действия. Например,

он обрабатывает вызовы `printf` таким образом, чтобы они отображались в текстовом поле на веб-странице и в окне консоли браузера, а не только в окне консоли.

Применяя HTML-шаблон, важно не указывать собственный объект `Module`, поскольку в этом случае все настройки шаблона будут удалены. При использовании HTML-шаблона любые значения, которые вы хотите установить, необходимо задавать непосредственно в существующем объекте `Module`, а не создавать новый.

В папке Chapter 7\7.2.3 dynamicLibraries\source\ создайте файл `pre.js` и затем откройте его в своем любимом редакторе. Нужно будет добавить массив, содержащий имя связываемого вспомогательного модуля, в свойство `dynamicLibraries` объекта `Module`. Добавьте следующий фрагмент кода в файл `pre.js`:

```
Module['dynamicLibraries'] = ['calculate_primes.wasm'];
```

После того как нужный JavaScript-код будет написан, можно перейти к последнему шагу процесса (рис. 7.14) и дать Emscripten команду сгенерировать модуль WebAssembly.

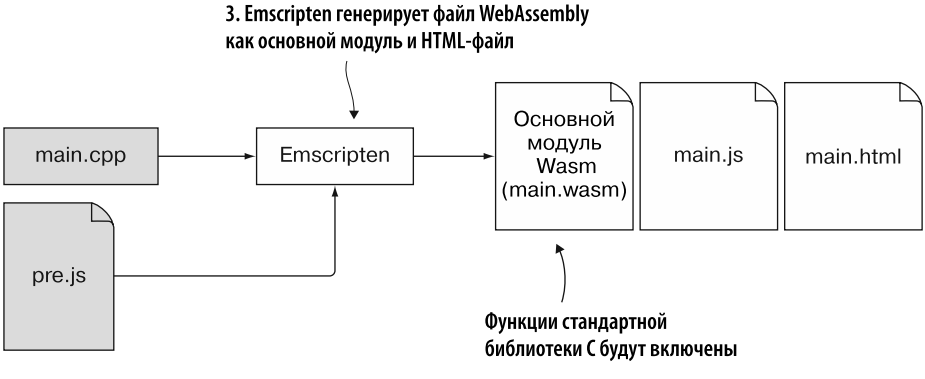


Рис. 7.14. Последний шаг процесса для реализации динамического связывания с использованием dynamicLibraries — командуем Emscripten сгенерировать модуль WebAssembly

Использование Emscripten для генерации файла WebAssembly как основного модуля из main.cpp

При использовании Emscripten для создания модуля WebAssembly необходимо, чтобы он включал содержимое файла `pre.js` в сгенерированный файл JavaScript. Чтобы Emscripten добавил файл, укажите его с помощью параметра командной строки `--pre-js`.

СОВЕТ

Имя файла pre.js используется здесь как пример соглашения об именовании, поскольку будет передано компилятору Emscripten через флаг --pre-js. Вам не обязательно задействовать это соглашение об именовании, но оно упрощает понимание назначения файла, когда вы обнаруживаете его в файловой системе.

Чтобы сгенерировать модуль WebAssembly в качестве основного модуля, откройте командную строку, перейдите к папке Chapter 7\7.2.3 dynamicLibraries\source\ и выполните следующую команду:

```
emcc main.cpp -s MAIN_MODULE=1 --pre-js pre.js
➥ -s EXPORTED_FUNCTIONS=['_putchar','_main'] -o main.html
```

После того как будет создан основной модуль WebAssembly, можно просмотреть результаты.

Просмотр результатов

Чтобы увидеть новый модуль WebAssembly в действии, откройте браузер и введите `http://localhost:8080/main.html` в адресную строку, чтобы перейти к сгенерированной веб-странице, показанной на рис. 7.15.

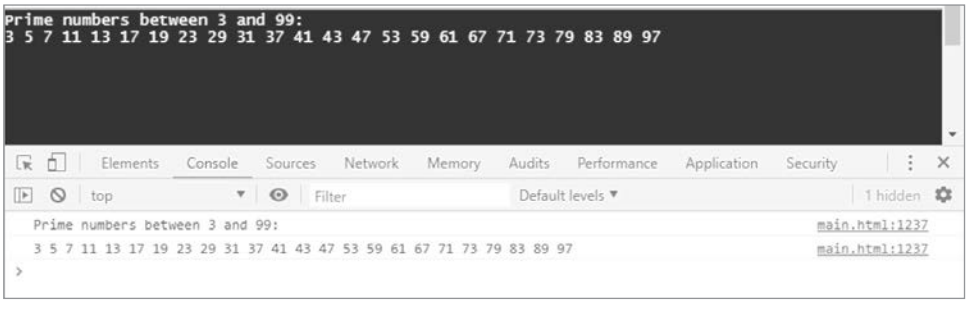


Рис. 7.15. Простые числа, определенные вспомогательным модулем, в случае когда оба модуля были связаны с помощью массива dynamicLibraries в Emscripten

Теперь представьте, что, завершив работу над модулем WebAssembly, в котором использовался подход `dynamicLibraries`, вы задались вопросом, может ли ваш начальник заинтересоваться еще и тем, как работает динамическое связывание вручную.

7.2.4. Динамическое связывание: JavaScript API в WebAssembly

При использовании `dlopen` вам нужно загрузить вспомогательный модуль, но после этого функция `dlopen` выполнит его связывание самостоятельно. При

использовании `dynamicLibraries` Emscripten самостоятельно выполняет загрузку и создание экземпляров модулей. При новом подходе нужно написать код JavaScript для загрузки и создания экземпляров модулей самостоятельно с применением JavaScript API в WebAssembly.

Для этого сценария вы решили взять файл `calculate_primes.c` из главы 3 и разделить его на две части, где один модуль WebAssembly будет содержать функцию `IsPrime`, а другой — функцию `FindPrimes`. Поскольку нам нужно использовать JavaScript API в WebAssembly, оба модуля WebAssembly необходимо будет скомпилировать как вспомогательные. Это значит, что ни один из них не будет иметь доступа к функциям стандартной библиотеки C. Без нее потребуется заменить вызовы `printf` вызовом собственной функции JavaScript, чтобы выводить простые числа в окно консоли браузера.

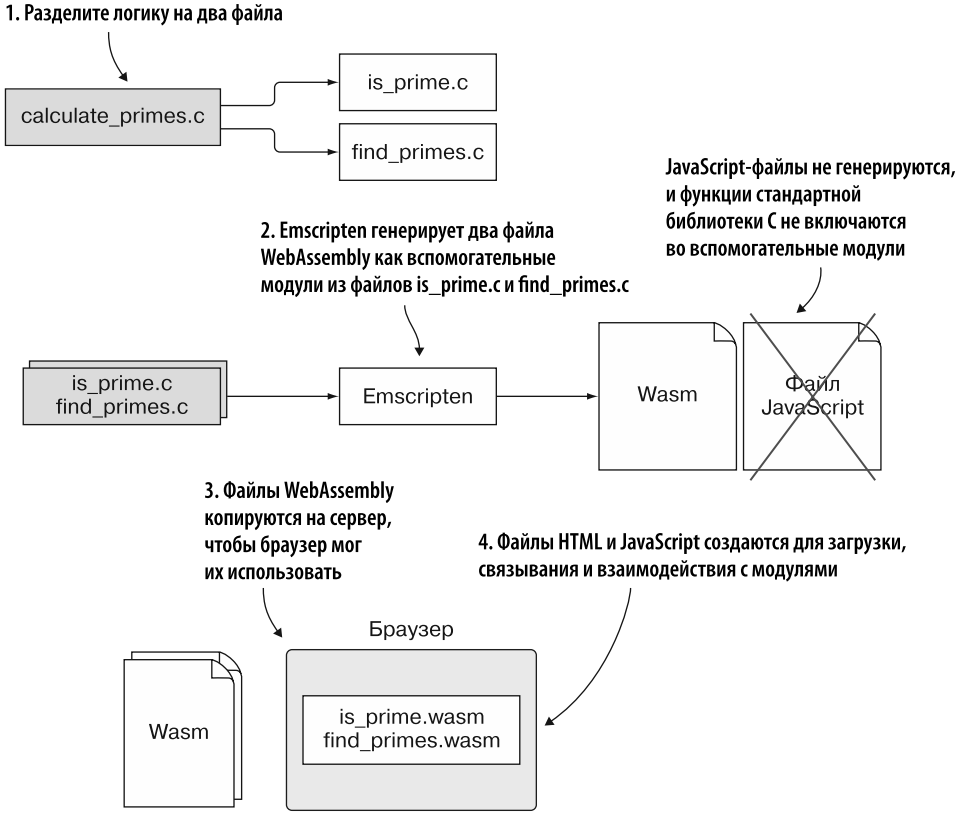


Рис. 7.16. Шаги для изменения файла calculate_primes.c, чтобы его можно было скомпилировать в два вспомогательных модуля WebAssembly. Сгенерированные файлы WebAssembly копируются на сервер, а затем создаются файлы HTML и JavaScript, требуемые для загрузки, связывания и взаимодействия с двумя модулями WebAssembly

Как показано на рис. 7.16, шаги для этого сценария будут следующими.

1. Разделите логику в `calculate_primes.c` на два файла: `is_prime.c` и `find_primes.c`.
2. Используйте Emscripten для создания дополнительных модулей WebAssembly из файлов `is_prime.c` и `find_primes.c`.
3. Скопируйте сгенерированные файлы WebAssembly на сервер, чтобы браузер мог их использовать.
4. Создайте файлы HTML и JavaScript, необходимые для загрузки, связывания и взаимодействия с двумя модулями WebAssembly с помощью JavaScript API в WebAssembly.

Разделение логики в файле calculate_primes.c на два файла

Как показано на рис. 7.17, первое, что нужно сделать, — копию файла `calculate_primes.c`, чтобы можно было настроить логику и разделить файл на две части. В папке Chapter 7\ создайте папку 7.2.4 ManualLinking\source\:

- скопируйте файл `calculate_primes.cpp` из папки Chapter 7\7.2.2 dlopen\source\ в новую папку source\. Переименуйте файл `calculate_primes.cpp`, который вы только что скопировали, в `is_prime.c`;
- сделайте копию файла `is_prime.c` и назовите его `find_primes.c`.

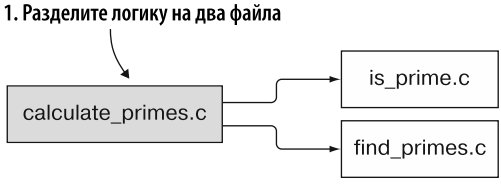


Рис. 7.17. Шаг 1 для реализации динамического связывания вручную с помощью JavaScript API в WebAssembly — изменение файла calculate_primes.c так, чтобы его логика была разделена на два файла

Откройте файл `is_prime.c` в своем любимом редакторе, а затем удалите следующие элементы:

- заголовочные файлы `cstdlib` и `cstdio`;
- открывающий блок `extern "C"` и закрывающую фигурную скобку в конце файла;
- функции `FindPrimes` и `main`, чтобы `IsPrime` осталась единственной функцией в файле.

Добавьте объявление `EMSCRIPTEN_KEEPALIVE` перед функцией `IsPrime`, чтобы функция `IsPrime` была включена в экспортируемые функции модуля. Откройте файл `find_primes.c` с помощью вашего любимого редактора и удалите следующие элементы:

- заголовочные файлы `cstdlib` и `cstdio`;
- открывающий блок `extern "C"` и закрывающую фигурную скобку в конце файла;
- функции `IsPrime` и `main`, чтобы `FindPrimes` осталась единственной функцией в файле.

Функция `FindPrimes` будет вызывать функцию `IsPrime` из модуля `is_prime`. Поскольку функция находится в другом модуле, необходимо включить сигнатуру функции `IsPrime`, которой предшествует ключевое слово `extern`, чтобы компилятор Emscripten знал, что функция будет доступна при запуске кода. Добавьте следующий фрагмент кода перед функцией `FindPrimes` в файле `find_primes.c`:

```
extern int IsPrime(int value);
```

Сейчас мы изменим функцию `FindPrimes` так, чтобы она вызывала в своем JavaScript-коде функцию `LogPrime`, а не `printf`. Поскольку эта функция вдобавок является внешней по отношению к модулю, также потребуется включить для нее сигнатуру функции. Добавьте следующий фрагмент кода перед сигнатурой функции `IsPrime` в файл `find_primes.c`:

```
extern void LogPrime(int prime);
```

Наконец, последнее, что нужно изменить в файле `find_primes.c`, — функцию `FindPrimes`, чтобы она больше не вызывала функцию `printf`. Удалите вызовы `printf` в начале и в конце функции; замените вызов `printf` внутри оператора `IsPrime if` на вызов функции `LogPrime`, но не передавайте строку. Передайте в функцию `LogPrime` только переменную `i`.

Измененная функция `FindPrimes` должна выглядеть следующим образом в файле `find_primes.c`:

```
EMSCRIPTEN_KEEPALIVE
void FindPrimes(int start, int end) {
  for (int i = start; i <= end; i += 2) {
    if (IsPrime(i)) {
      LogPrime(i);   <-- printf заменяется вызовом LogPrime
    }
  }
}
```

После того как код на C будет создан, можно перейти к шагу 2 (рис. 7.18), который заключается в использовании Emscripten для компиляции кода во вспомогательные модули WebAssembly.

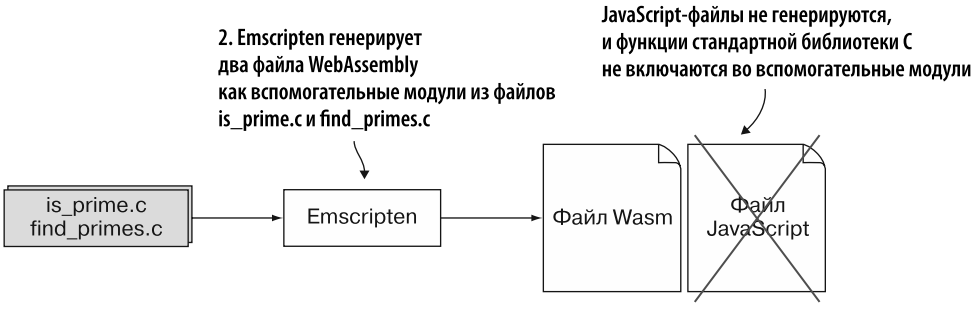


Рис. 7.18. Шаг 2 — использование Emscripten для генерации вспомогательных модулей WebAssembly из двух созданных файлов

Использование Emscripten для генерации вспомогательных модулей WebAssembly

Чтобы сгенерировать модуль WebAssembly из файла `is_prime.c`, откройте командную строку, перейдите в папку 7.2.4 ManualLinking\source\, а затем выполните следующую команду:

```
emcc is_prime.c -s SIDE_MODULE=2 -O1 -o is_prime.wasm
```

Чтобы сгенерировать модуль WebAssembly из файла `find_primes.c`, выполните следующую команду:

```
emcc find_primes.c -s SIDE_MODULE=2 -O1 -o find_primes.wasm
```

После того как два модуля WebAssembly будут созданы, можно сделать следующие шаги — создать веб-страницы и файлы JavaScript для загрузки, связывания и взаимодействия с модулями (рис. 7.19).

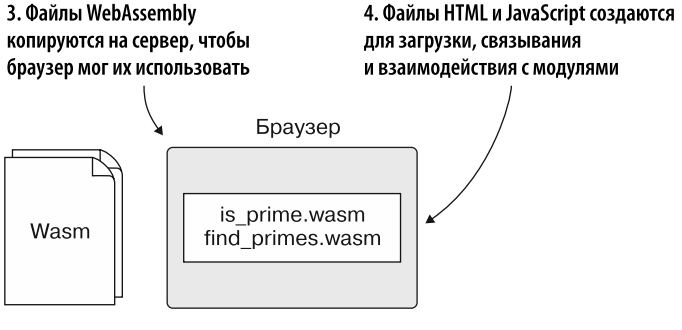


Рис. 7.19. Последние шаги — создание файлов HTML и JavaScript для загрузки, связывания и взаимодействия с модулями WebAssembly

Создание файлов HTML и JavaScript

В папке Chapter 7\7.2.4 ManualLinking\ создайте папку frontend\:

- скопируйте файлы `is_prime.wasm` и `find_primes.wasm` из папки 7.2.4 ManualLinking\source\ в новую папку frontend\;
- создайте файл `main.html` в папке frontend\, а затем откройте его в своем любимом редакторе.

HTML-файл будет представлять очень простую веб-страницу. В нем будет текст, чтобы вы знали, что страница загружена, а затем тег `script` для загрузки в файл JavaScript (`main.js`), который будет обрабатывать загрузку и связывание двух вспомогательных модулей.

Добавьте содержимое листинга 7.5 в файл `main.html`.

Листинг 7.5. Содержимое файла main.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    HTML page I created for my WebAssembly module.

    <script src="main.js"></script>
  </body>
</html>
```

Следующий шаг — создание JavaScript-файла, который будет обрабатывать загрузку и связывание двух модулей WebAssembly. В папке 7.2.4 ManualLinking\frontend\ создайте файл `main.js`, а затем откройте его в редакторе.

Модуль `find_primes` в WebAssembly будет ожидать функцию, которую может вызвать для передачи простого числа в JavaScript-код. Создадим функцию `logPrime`, которая будет передаваться модулю во время создания экземпляра и записывать полученное от модуля значение в окно консоли инструментов разработчика в браузере.

Добавьте следующий фрагмент кода в файл `main.js`:

```
function logPrime(prime) {
  console.log(prime.toString());
}
```

WebAssembly-модуль `find_primes` зависит от функции `IsPrime` в модуле `is_prime`, поэтому необходимо сначала загрузить и создать экземпляр модуля `is_prime`. В методе `then` вызова `instantiateStreaming` для модуля `is_prime`:

- создайте `importObject` для WebAssembly-модуля `find_primes`. Этому `importObject` будет предоставлена экспортированная функция `_IsPrime` из модуля `is_prime`, а также JavaScript-функция `logPrime`;
- вызовите функцию `instantiateStreaming` для WebAssembly-модуля `find_primes` и верните `Promise`.

Следующий метод `then` будет использоваться для успешной загрузки и создания экземпляра WebAssembly-модуля `find_primes`. В этом блоке вы вызовете функцию `_FindPrimes`, передав диапазон значений, чтобы простые числа в данном диапазоне показывались в окне консоли браузера.

Добавьте код, показанный в листинге 7.6, в файл `main.js` после функции `logPrime`.

Листинг 7.6. Загрузка и связывание двух модулей WebAssembly

```
...
const isPrimeImportObject = {
  env: {
    __memory_base: 0,
  }
};

WebAssembly.instantiateStreaming(fetch("is_prime.wasm"),
    isPrimeImportObject)
.then(module => {

  const findPrimesImportObject = {
    env: {
      __memory_base: 0,
      _IsPrime: module.instance.exports._IsPrime,
      _LogPrime: logPrime,
    }
  };

  return WebAssembly.instantiateStreaming(fetch("find_primes.wasm"),
      findPrimesImportObject);

})
.then(module => {
  module.instance.exports._FindPrimes(3, 100);
});
```

importObject для модуля is_prime

Загружает и создает экземпляр модуля is_prime

Модуль is_prime готов

importObject для модуля find_primes

Экспортированная функция передается в модуль find_primes

JavaScript-функция передается в модуль find_primes

Модуль find_primes готов

Загружает и создает экземпляр модуля find_primes. Возвращает экземпляр модуля

Выводит простые числа между 3 и 100 в окно консоли

Просмотр результатов

Создав HTML- и JavaScript-код, вы можете открыть браузер и ввести `http://localhost:8080/main.html` в адресную строку, чтобы просмотреть веб-страницу. Нажмите `F12`, чтобы открыть окно консоли инструментов разработчика браузера. Вы должны увидеть простые числа от 3 до 100, как показано на рис. 7.20.

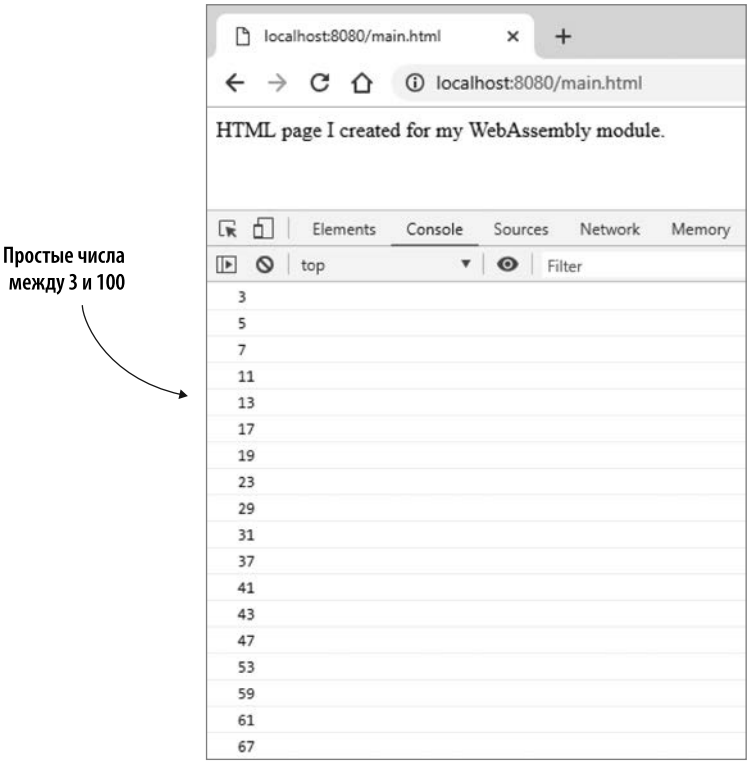


Рис. 7.20. Простые числа между 3 и 100, показанные WebAssembly-модулем find_primes

Узнав, как реализовать динамическое связывание, используя все три подхода, пора перейти к их сравнению.

7.3. ОБЗОР ДИНАМИЧЕСКОГО СВЯЗЫВАНИЯ

В этой главе вы узнали о трех подходах к динамическому связыванию:

- `dlopen`:
  - сначала необходимо загрузить вспомогательный модуль в файловую систему Emscripten;
  - вызов `dlopen` возвращает дескриптор файла вспомогательного модуля;
  - передача дескриптора и имени функции, которую нужно вызвать, в функцию `dlsym` вернет указатель на функцию во вспомогательном модуле;
  - на данном этапе вызов указателя функции аналогичен вызову обычной функции в модуле;

  - поскольку вы запрашиваете имя функции на основе дескриптора вспомогательного модуля, наличие функции с таким же именем в основном модуле не вызовет никаких проблем;
  - привязка к вспомогательному модулю выполняется по мере необходимости;
- `dynamicLibraries`:
  - вы предоставляете Emscripten список вспомогательных модулей, с которыми нужно его связать, включив их в свойство массива `dynamicLibraries` объекта `Module`. Этот список необходимо указать до инициализации кода JavaScript в Emscripten;
  - Emscripten выполняет загрузку и связывание вспомогательного модуля с основным самостоятельно;
  - код основного модуля вызывает функции во вспомогательном модуле так же, как он вызывает собственные функции;
  - невозможно вызвать функцию в другом модуле, если уже существует функция с таким именем в текущем модуле;
  - все указанные вспомогательные модули связываются сразу после инициализации JavaScript-кода в Emscripten;
- JavaScript API в WebAssembly:
  - загрузка модуля WebAssembly обрабатывается с помощью метода `fetch` и используется JavaScript API в WebAssembly для создания экземпляра этого модуля;
  - затем вы загружаете следующий модуль WebAssembly и передаете необходимые экспортируемые данные из первого модуля в качестве импорта для текущего модуля;
  - код основного модуля вызывает функции во вспомогательном модуле так же, как он вызывает собственные функции;
  - как и при подходе `dynamicLibraries`, невозможно вызвать функцию в другом модуле, если уже существует функция с таким именем в текущем модуле.

Таким образом, выбор подхода к динамическому связыванию действительно зависит от того, какой контроль над процессом вы хотите получить и нужен ли этот контроль в модуле или в JavaScript:

- `dlopen` предоставляет управление динамическим связыванием серверному коду. Это также единственный возможный подход, если нужно вызвать функцию во вспомогательном модуле, когда уже существует функция с таким именем в основном модуле;
- `dynamicLibraries` передает возможность управления динамическим связыванием инструментарию, и Emscripten выполняет всю работу за вас;

- JavaScript API в WebAssembly предоставляет управление динамическим связыванием коду внешнего интерфейса, где связывание обрабатывается с помощью JavaScript.

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже приведены некоторые возможные варианты использования того, что вы узнали в этой главе.

- Игровой движок — то, что может ощутить преимущества динамического связывания. При скачивании игры движок тоже нужно сначала загрузить, а затем кэшировать. В следующий раз, когда вы начнете играть, фреймворк сможет проверить, установлен ли уже движок в системе, и если да, то загрузить только запрошенную игру. Это сэкономит время и пропускную способность.
- Вы можете создать модуль редактирования изображений так, чтобы основная логика загружалась сразу, но функции, которые могут использоваться не так часто (например, определенные фильтры), могли загружаться по запросу.
- Вы можете представить веб-приложение с несколькими уровнями подписки. На бесплатном уровне будет меньше всего функций, поэтому загрузится только базовый модуль. Для уровня «Премиум» может быть включена дополнительная логика. Например, уровень «Премиум» вашего веб-приложения добавляет возможность отслеживать расходы. Дополнительный модуль можно использовать для анализа содержимого файла Excel и его форматирования так, как нужно серверному коду.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Используя один из методов динамического связывания, которые вы изучили в этой главе, создайте следующее:

   а) вспомогательный модуль, содержащий функцию `Add`, которая принимает два целочисленных параметра и возвращает сумму как целое число;

   б) основной модуль, который имеет функцию `main()`, вызывающую функцию `Add` вспомогательного модуля, и отображает результат в окне консоли инструментов разработчика браузера.

2. Какой подход динамического связывания вы бы использовали, если бы вам нужно было вызвать функцию во вспомогательном модуле, но она называлась бы так же, как функция в основном?

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Как и в большинстве случаев, у использования динамического связывания есть свои плюсы и минусы. Прежде чем применять конкретный подход, вы должны решить, перевешивают ли его преимущества недостатки для вашего приложения.
- При необходимости динамическое связывание может выполняться кодом WebAssembly с помощью функции `dlopen`.
- Можно указать сгенерированному Emscripten JavaScript, что требуется связь определенных вспомогательных модулей с основным. Emscripten автоматически свяжет модули во время создания экземпляра.
- Используя JavaScript API в WebAssembly, можно вручную загружать, создавать экземпляры и связывать несколько вспомогательных модулей.
- Вы можете управлять выполнением сгенерированного Emscripten кода JavaScript, создав объект `Module` перед включением файла JavaScript в Emscripten. Вы также можете настроить объект `Module`, добавив собственный JavaScript-код в файл JavaScript, созданный Emscripten, используя флаг командной строки `--pre-js` при компиляции модуля WebAssembly.

8 Динамическое связывание: реализация

В этой главе

- ✓ Использование динамического связывания в одностраничном приложении.
- ✓ Создание нескольких экземпляров объекта `Module` из Emscripten JavaScript, каждый экземпляр которого динамически связан с другим вспомогательным модулем WebAssembly.
- ✓ Уменьшение размера основного модуля WebAssembly за счет включения флага удаления неиспользуемого кода.

В главе 7 вы узнали о различных подходах к динамическому связыванию модулей WebAssembly:

- `dlopen`, при котором код на C или C++ вручную связывается с модулем, получая указатели на определенные функции по мере необходимости;
- `dynamicLibraries`, при котором JavaScript предоставляет Emscripten список модулей для связывания, и Emscripten автоматически связывается с этими модулями во время инициализации;
- связывание вручную, при котором JavaScript берет экспорт одного модуля и передает его как импорт в другой модуль с помощью JavaScript API в WebAssembly.

В этой главе мы собираемся использовать подход `dynamicLibraries`, при котором Emscripten выполняет динамическое связывание самостоятельно на основе указанного списка модулей.

Предположим, что компания, создавшая онлайн-версию страницы Edit Product (Изменить товар) приложения интернет-магазина, теперь хочет создать форму Place Order (Разместить заказ), показанную на рис. 8.1. Как и страница Edit Product (Изменить товар), форма Place Order (Разместить заказ) будет применять модуль WebAssembly для проверки введенных пользователем данных.

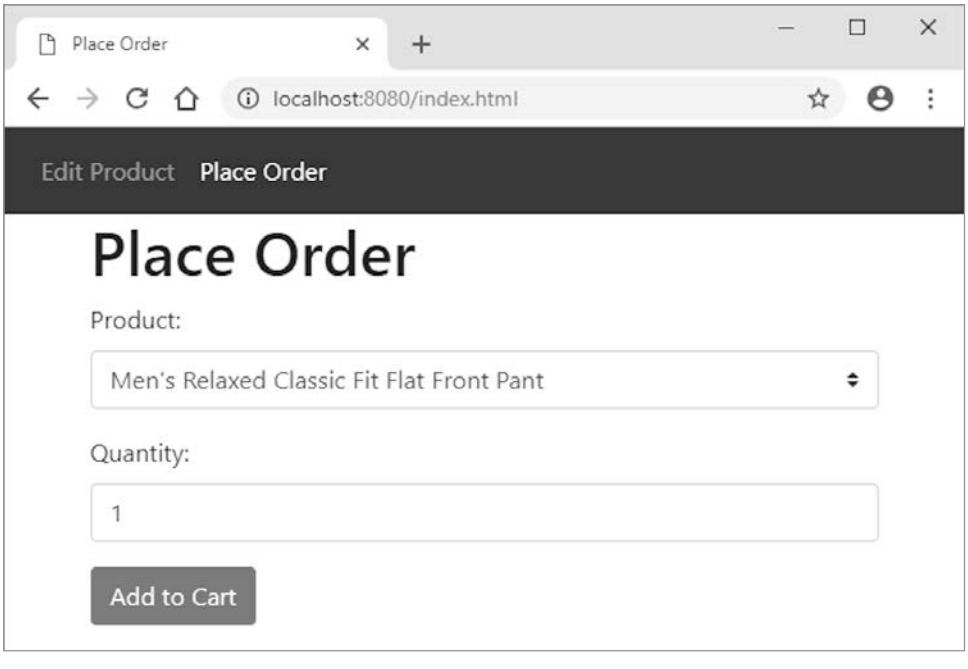


Рис. 8.1. Новая форма Place Order (Разместить заказ)

В процессе планирования того, как будет работать новая веб-страница, компания замечает, что ей потребуется проверка, аналогичная проверке на существующей странице Edit Product (Изменить товар):

- обе страницы требуют, чтобы из раскрывающегося списка можно было выбрать правильный элемент;
- обе страницы должны гарантировать, что значение было указано.

Вместо того чтобы дублировать логику, указанную в модуле WebAssembly для каждой страницы, компания хотела бы взять общую логику — проверку того, было

ли указано значение, и того, находится ли выбранный идентификатор в массиве действительных идентификаторов, — и переместить ее в отдельный модуль. Затем модуль проверки каждой страницы будет динамически связан во время выполнения с модулем, хранящим общую логику для получения доступа к основным функциям, которые ему нужны, как показано на рис. 8.2. Несмотря на то что два отдельных модуля будут просто вызывать друг друга по мере необходимости, касательно кода создается впечатление, что вы работаете только с одним модулем.

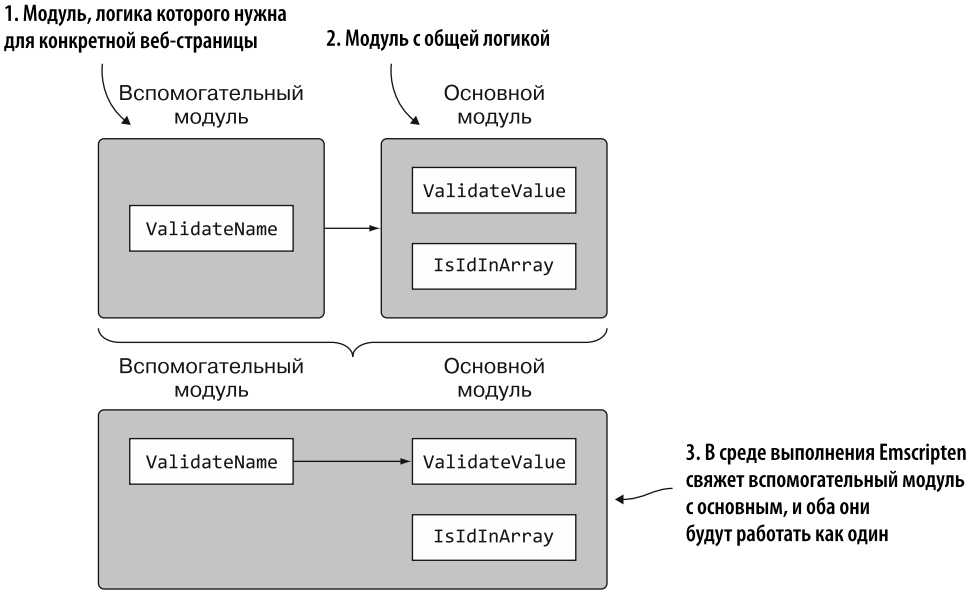


Рис. 8.2. Во время выполнения логика, необходимая для конкретной страницы (вспомогательный модуль), будет связана с общей логикой (основной модуль). Со стороны кода два модуля будут работать как один

Для этого сценария компания хотела бы настроить сайт так, чтобы он работал как *одностраничное приложение* (single-page application, SPA).

ОПРЕДЕЛЕНИЕ

Что такое SPA? На традиционном сайте для каждой веб-страницы существует отдельный HTML-файл. В случае с SPA вы используете только одну HTML-страницу, и ее содержимое изменяется с помощью кода, который выполняется в браузере, в зависимости от действий пользователя.

Настройка веб-страницы для работы в качестве SPA добавляет несколько интересных особенностей, когда дело доходит до динамического связывания с подходом

`dynamicLibraries`; вы указываете все вспомогательные модули, которые Emscripten должен связать, до инициализации JavaScript в Emscripten. Обычно созданный Emscripten код JavaScript существует как глобальный объект `Module` и инициализируется в момент загрузки браузером файла JavaScript. Когда JavaScript в Emscripten инициализируется, все указанные вспомогательные модули связываются с основным.

Одно из преимуществ динамического связывания — загрузка модуля и связывание с ним только по мере необходимости, что сокращает время загрузки и обработки при первой загрузке страницы. При работе с SPA вам нужно указать лишь вспомогательный модуль для страницы, которая отображается изначально. Когда пользователь переходит на следующую страницу, как указать нужный вспомогательный модуль в SPA, если объект `Module` в Emscripten уже инициализирован?

Оказывается, есть параметр, который можно указать при компиляции основного модуля (`-s MODULARIZE=1`); он сообщит компилятору Emscripten о необходимости обернуть объект `Module` сгенерированного Emscripte файла JavaScript в функцию. Таким образом решаются две проблемы:

- теперь момент инициализации объекта `Module` находится под вашим контролем — вам нужно самим создавать экземпляр объекта для его использования;
- поскольку вы самостоятельно создаете экземпляр объекта `Module`, можно не ограничиваться одним экземпляром. Это позволит вам создать второй экземпляр основного модуля WebAssembly и иметь ссылку на данный экземпляр во вспомогательном модуле, относящемся ко второй странице.

8.1. СОЗДАНИЕ МОДУЛЕЙ WEBASSEMBLY

В главах 3–5 вы создавали модули как вспомогательные, чтобы не создавать файл JavaScript в Emscripten; это позволяет вручную обрабатывать загрузку модуля и создание его экземпляра с помощью JavaScript API в WebAssembly. Данный побочный эффект полезен, однако в действительности вспомогательные модули предназначены для динамического связывания, для чего и будут использоваться в текущей главе.

Во вспомогательных модулях нет файла JavaScript, созданного Emscripten, или функций стандартной библиотеки C, поскольку они связываются с основным модулем в среде выполнения. Эти функции хранятся в основном модуле, и после связывания вспомогательные модули получают к ним доступ.

НАПОМИНАНИЕ

При динамическом связывании может быть несколько вспомогательных модулей, связанных с основным, но основной модуль может быть только один.

На рис. 8.3 показаны следующие шаги для изменения кода на C++ и создания модулей WebAssembly.

1. Разделите логику в файле `validate.cpp` на два файла: один для общей логики, которая будет совместно использоваться (`validate_core.cpp`), и один для логики, необходимой для страницы Edit Product (Изменить товар) (`validate_product.cpp`).
2. Создайте новый файл C++ для логики, которая будет использоваться в новой форме Place Order (Разместить заказ) (`validate_order.cpp`).

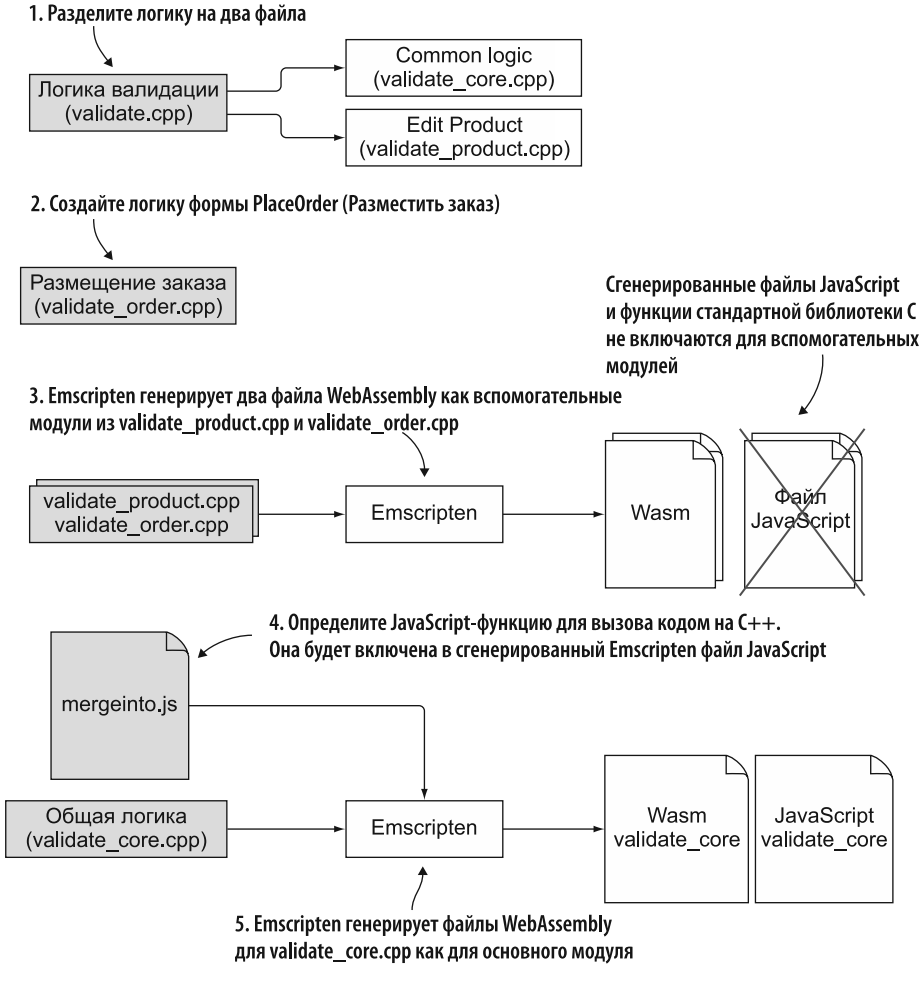


Рис. 8.3. Шаги, необходимые для изменения логики на C++ и генерации модулей WebAssembly

3. Используйте Emscripten для создания дополнительных модулей WebAssembly из `validate_product.cpp` и `validate_order.cpp`.
4. Определите JavaScript-функцию, которую код на C++ будет вызывать в случае возникновения проблем с проверкой. Функция будет помещена в файл `mergeinto.js` и включена в созданный Emscripten файл JavaScript во время компиляции основного модуля.
5. Emscripten будет использоваться для создания файла WebAssembly в качестве основного модуля из `validate_core.cpp`.

   После того как модули WebAssembly будут созданы, остается сделать следующие шаги для настройки сайта (рис. 8.4).

6. Измените веб-страницу так, чтобы на ней были панель навигации и элементы управления формы Place Order (Разместить заказ). Затем измените JavaScript, чтобы отобразить правильный набор элементов управления в зависимости от того, какая ссылка навигации была нажата.
7. Измените JavaScript веб-страницы, чтобы связать нужный вспомогательный модуль с модулем общей логики. Добавьте также код JavaScript для проверки формы Place Order (Разместить заказ).

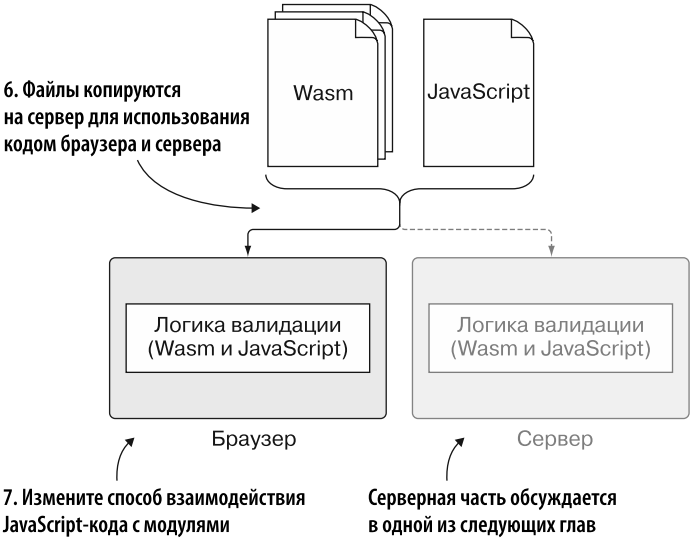


Рис. 8.4. Шаги по изменению HTML-кода для создания формы Place Order (Разместить заказ) и по изменению кода на JavaScript для реализации динамического связывания модулей WebAssembly в браузере и серверного кода. Серверная часть, Node.js, обсуждается в одной из следующих глав

8.1.1. Разделение логики в validate.cpp на два файла

Как показано на рис. 8.5, первый шаг — изменение кода на C++, написанного в главе 5, чтобы логика, которая будет использоваться формами Edit Product (Изменить товар) и Place Order (Разместить заказ), располагалась в отдельных файлах. Логика, относящаяся к форме Edit Product (Изменить товар), будет перемещена в новый файл.

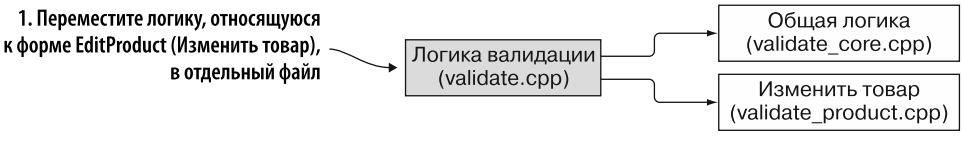


Рис. 8.5. Шаг 1 процесса — перемещение логики формы Edit Product (Изменить товар) в отдельный файл

В папке WebAssembly создайте папку Chapter 8\8.1 EmDynamicLibraries\source\ для файлов, которые будут использоваться в этом подразделе, а затем выполните следующие действия:

- скопируйте файл `validate.cpp` из библиотеки Chapter 5\5.1.1 EmJsLibrary\source\ folder в новую исходную папку;
- сделайте копию файла `validate.cpp` и переименуйте его в `validate_product.cpp`;
- переименуйте другую копию файла `validate.cpp` в `validate_core.cpp`.

Первое, что нужно сделать, — удалить логику изменения товара из файла `validate_core.cpp`, поскольку этот файл будет использоваться для создания общего модуля WebAssembly, который пригодится для обеих форм: и Edit Product (Изменить товар), и Place Order (Разместить заказ).

Изменение файла validate_core.cpp

Откройте файл `validate_core.cpp` в своем любимом редакторе, а затем удалите функции `ValidateName` и `ValidateCategory`. Удалите строку включения `cstring`, поскольку она больше не нужна в этом файле.

Поскольку функции `ValidateValueProvided` и `IsCategoryIdInArray` теперь будут вызываться другими модулями, эти функции необходимо будет экспортировать. Добавьте следующий фрагмент кода перед функциями `ValidateValueProvided` и `IsCategoryIdInArray` в файле `validate_core.cpp`:

```
#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
```

Можно задействовать функцию `IsCategoryIdInArray`, чтобы проверить, входит ли идентификатор в какой-либо указанный массив, но имя, которое использует функция, указывает, что эта функция применяется только для идентификатора категории. Нужно изменить данную функцию так, чтобы ее имя было более общим, поскольку она будет использоваться обоими вспомогательными модулями.

Измените функцию `IsCategoryIdInArray` в файле `validate_core.cpp` так, чтобы слово «категория» (`category`) больше не использовалось. Функция должна соответствовать коду, показанному в листинге 8.1.

Листинг 8.1. Функция IsCategoryIdInArray изменена и переименована в IsIdInArray

```
...
#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
int IsIdInArray(char* selected_id, int* valid_ids, int array_length) {
  int id = atoi(selected_id);
  for (int index = 0; index < array_length; index++) {
    if (valid_ids[index] == id) {
      return 1;
    }
  }

  return 0;
}
...
```

Автоматически добавляет функцию IsIdInArray в список экспортируемых модулем функций

После того как логика страницы Edit Product (Изменить товар) удалена из файла `validate_core.cpp` и функция `IsCategoryIdInArray` изменена на более общую, необходимо пересмотреть логику данной страницы.

Настройка файла validate_product.cpp

Откройте файл `validate_product.cpp` в своем любимом редакторе и удалите функции `ValidateValueProvided` и `IsCategoryIdInArray`, так как они теперь являются частью модуля `validate_core`. Поскольку функции `ValidateValueProvided` и `IsIdInArray` теперь являются частью другого модуля, нужно будет включить их сигнатуры функций с префиксом `extern`, чтобы компилятор знал, что функции будут доступны при запуске кода.

Добавьте следующие сигнатуры функций в блок `extern "C"` перед сигнатурой функции `extern UpdateHostAboutError` в файле `validate_product.cpp`:

```
extern int ValidateValueProvided(const char* value,
    const char* error_message);

extern int IsIdInArray(char* selected_id, int* valid_ids,
    int array_length);
```

Поскольку вы переименовали `IsCategoryIdInArray` в `IsIdInArray` в основной логике, необходимо изменить функцию `ValidateCategory`, чтобы вместо нее вызывать `IsIdInArray`. Функция `ValidateCategory` в файле `validate_product.cpp` должна соответствовать коду, показанному в листинге 8.2.

Листинг 8.2. Измененная функция ValidateCategory (файл validate_product.cpp)

```
...

int ValidateCategory(char* category_id, int* valid_category_ids,
    int array_length) {
  if (ValidateValueProvided(category_id,
      "A Product Category must be selected.") == 0) {
    return 0;
  }

  if ((valid_category_ids == NULL) || (array_length == 0)) {
    UpdateHostAboutError("There are no Product Categories available.");
    return 0;
  }

  if (IsIdInArray(category_id, valid_category_ids,
      array_length) == 0) {
    UpdateHostAboutError("The selected Product Category is not valid.");
    return 0;
  }

  return 1;
}
...
```

Функция переименована в IsIdInArray

После того как логика страницы Edit Product (Изменить товар) будет отделена от общей логики, можно сделать следующий шаг — создать логику формы Place Order (Разместить заказ) (рис. 8.6).

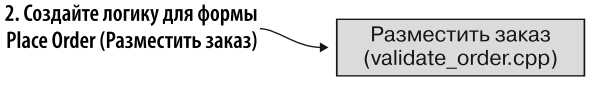


Рис. 8.6. Шаг 2 процесса — создание логики для формы Place Order (Разместить заказ)

8.1.2. Создание нового файла C++ для логики формы Place Order (Разместить заказ)

В папке Chapter 8\8.1 EmDynamicLibraries\source\ создайте файл `validate_order.cpp` и откройте его в своем любимом редакторе. При создании вспомогательного модуля в предыдущих главах вы не включали заголовочные файлы стандартной библиотеки C, так как используемые функции не будут доступны во время выполнения. В этом случае, поскольку вспомогательный модуль будет связан с основным (`validate_core`), а основной модуль будет иметь доступ к стандартной библиотеке C, вспомогательный модуль сможет получить доступ к этим функциям.

Добавьте строки включения для заголовочных файлов стандартной библиотеки C и Emscripten, а также блок `extern "C"` в файл `validate_order.cpp`, как показано в листинге 8.3.

Листинг 8.3. Заголовочные файлы и блок extern "C" добавлены в файл validate_order.cpp

```
#include <cstdlib>

#ifdef __EMSCRIPTEN__
  #include <emscripten.h>
#endif

#ifdef __cplusplus
extern "C" {
#endif
          ◄──┘ Здесь будут добавлены ваши функции WebAssembly

#ifdef __cplusplus
}
#endif
```

Вам нужно будет добавить сигнатуры для функций `ValidateValueProvided` и `IsIdInArray`, которые находятся в модуле `validate_core`. Кроме того, вы добавите сигнатуру функции `UpdateHostAboutError`, которую модуль будет импортировать из кода JavaScript.

Добавьте сигнатуры функций, показанные ниже, в блок `extern "C"` в файле `validate_order.cpp`:

```
extern int ValidateValueProvided(const char* value,
    const char* error_message);

extern int IsIdInArray(char* selected_id, int* valid_ids,
    int array_length);

extern void UpdateHostAboutError(const char* error_message);
```

Будущая форма Place Order (Разместить заказ) будет иметь раскрывающийся список товаров и поле их количества, которые требуется проверить. Оба значения поля будут переданы в модуль в виде строк, но идентификатор товара будет содержать числовое значение.

Чтобы проверить выбранный пользователем идентификатор товара и введенное количество, создадим две функции: `ValidateProduct` и `ValidateQuantity`. Первая позволяет убедиться, что выбран действительный идентификатор товара.

Функция ValidateProduct

Функция `ValidateProduct` получит следующие параметры:

- выбранный пользователем идентификатор товара;
- указатель на массив целых чисел, содержащий допустимые идентификаторы товаров;
- количество элементов в массиве допустимых идентификаторов товаров.

Функция проверит три вещи:

- был ли предоставлен идентификатор товара;
- был ли предоставлен указатель на массив допустимых идентификаторов товаров;
- находится ли выбранный пользователем идентификатор товара в массиве допустимых идентификаторов.

Если какая-либо из проверок завершится неудачно, то в код JavaScript будет передано сообщение об ошибке с помощью вызова функции `UpdateHostAboutError`. Затем функция `ValidateProduct` вернет `0`, чтобы указать, что произошла ошибка. Если код выполняется до конца функции, проблем с проверкой не обнаружено, то возвращается сообщение `1` (успех).

Добавьте функцию `ValidateProduct`, показанную в листинге 8.4, под сигнатурой функции `UpdateHostAboutError` и внутри блока `extern "C"` в файле `validate_order.cpp`.

Листинг 8.4. Функция ValidateProduct

```
#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
int ValidateProduct(char* product_id, int* valid_product_ids,
    int array_length) {
  if (ValidateValueProvided(product_id,
      "A Product must be selected.") == 0) {
    return 0;
```

Если значение не получено, то возвращает ошибку

```
  }

  if ((valid_product_ids == NULL) || (array_length == 0)) {
    UpdateHostAboutError("There are no Products available.");
    return 0;
  }

  if (IsIdInArray(product_id, valid_product_ids,
      array_length) == 0) {
    UpdateHostAboutError("The selected Product is not valid.");
    return 0;
  }

  return 1;
}
```

Если массив не задан, то возвращает ошибку

Если идентификатор выбранного товара не найден в массиве, то возвращает ошибку

Сообщает, что все прошло успешно

Последняя функция, которую вам нужно будет создать, — `ValidateQuantity`. Она позволяет убедиться, что введенное пользователем значение количества допустимо.

Функция ValidateQuantity

Функция `ValidateQuantity` будет принимать единственный параметр — введенное пользователем количество — и возвращать результат двух проверок:

- было ли указано количество;
- значение количества больше либо равно 1?

Если какая-либо проверка не удалась, то в код JavaScript будет передано сообщение об ошибке через вызов функции `UpdateHostAboutError`, а сама функция вернет `0` (ноль), чтобы указать, что произошла ошибка. Если код выполняется до конца функции, проблем с проверкой не обнаружено, то возвращается сообщение `1` (успех).

Добавьте функцию `ValidateQuantity`, показанную в листинге 8.5, под функцией `ValidateProduct` и в блок `extern "C"` в файле `validate_order.cpp`.

Листинг 8.5. Функция ValidateQuantity

```
#ifdef __EMSCRIPTEN__
  EMSCRIPTEN_KEEPALIVE
#endif
int ValidateQuantity(char* quantity) {
  if (ValidateValueProvided(quantity,
      "A quantity must be provided.") == 0) {
    return 0;
  }

  if (atoi(quantity) <= 0) {
```

Если значение не получено, то вернется ошибка

Если значение меньше 1, то вернется ошибка

```
    UpdateHostAboutError("Please enter a valid quantity.");
    return 0;
  }

  return 1;
}
```

Указывает, что проверка прошла успешно

Код на C++ отредактирован. Следующая часть процесса (рис. 8.7) — компиляция файлов на C++ в модули WebAssembly с помощью Emscripten.

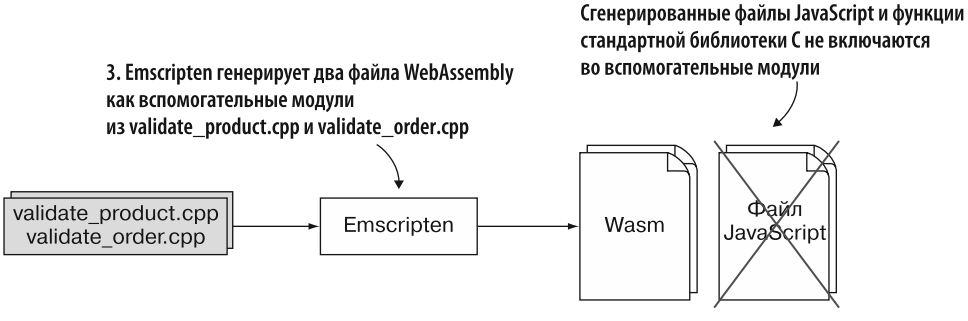


Рис. 8.7. Шаг 3 — использование Emscripten для компиляции файлов на C++ в модули WebAssembly

8.1.3. Использование Emscripten для генерации вспомогательных модулей WebAssembly

При использовании динамического связывания с Emscripten нельзя иметь более одного основного модуля. Он будет включать стандартные функции библиотеки C и созданный Emscripten файл JavaScript. Во вспомогательных модулях ни одна из этих функций не включается, но при связывании с основным модулем вспомогательные получают доступ к нужным функциям. Файл `validate_core.cpp` будет собран как основной модуль, а два других файла на C++, `validate_product.cpp` и `validate_order.cpp`, — как вспомогательные.

По умолчанию при создании основного модуля Emscripten будет включать все стандартные функции библиотеки C в модуль WebAssembly, поскольку не знает, какие из них потребуются вспомогательным модулям. Это значительно увеличивает размер модуля, особенно если вы используете только несколько стандартных функций библиотеки C.

Чтобы оптимизировать основной модуль, можно дать Emscripten указание включить только определенные стандартные функции библиотеки C. Мы будем использовать данный подход, но прежде, чем это сделать, необходимо знать,

какие функции включить. Чтобы определить это, вы всегда можете прочитать код построчно, но при этом пропустить некоторые функции. Другой подход — закомментировать заголовочные файлы для стандартной библиотеки C, а затем запустить командную строку с целью создать модуль WebAssembly. Компилятор Emscripten увидит, что для используемых стандартных функций библиотеки C не определены сигнатуры функций, и отобразит соответствующую ошибку.

Мы будем использовать второй подход, поэтому вспомогательные модули нужно скомпилировать до компиляции основного. Как показано на рис. 8.8, первый создаваемый модуль WebAssembly — вспомогательный модуль для страницы Edit Product (Изменить товар) (`validate_product.cpp`).

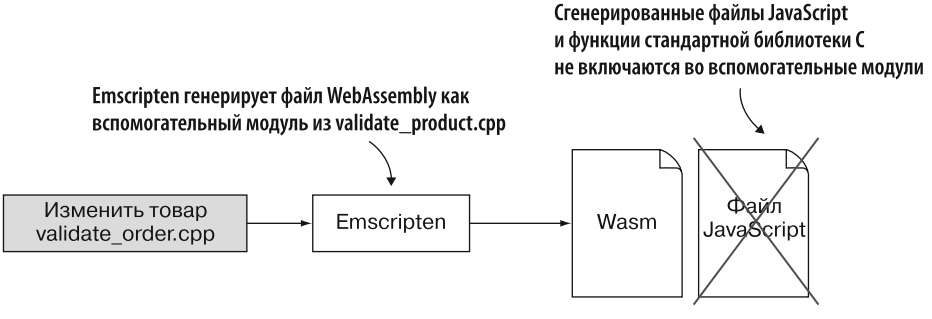


Рис. 8.8. Emscripten используется для генерации модуля WebAssembly, выполняющего проверку страницы Edit Product (Изменить товар)

Генерация вспомогательного модуля для изменения товара: validate_product.cpp

В предыдущих главах при создании вспомогательных модулей WebAssembly вы заменяли стандартные заголовочные файлы библиотеки C на заголовочный файл для некоего замещающего кода, созданного в главе 4. В этом случае замещающий код не нужен, поскольку вспомогательный модуль будет связан с основным в среде выполнения, а в основном будут находиться функции стандартной библиотеки C.

При компиляции основного модуля в подразделе 8.1.5 вы предоставляете Emscripten список стандартных функций библиотеки C, которые применяются во вспомогательных модулях. Чтобы определить, какие функции используются в коде, закомментируйте заголовочные файлы стандартной библиотеки C, а затем попытайтесь скомпилировать модуль. Если задействованы какие-либо функции стандартной библиотеки C, то компилятор Emscripten выдаст ошибки об отсутствующих определениях функций.

Но прежде, чем вы попытаетесь определить, какие стандартные функции библиотеки C используются, необходимо скомпилировать модуль обычным образом, чтобы убедиться в отсутствии проблем. Нужно быть уверенными, что ошибки

при комментировании заголовочных файлов связаны с отсутствующими определениями функций. Чтобы скомпилировать модуль в обычном режиме, откройте командную строку, перейдите в папку Chapter 8\8.1 EmDynamicLibraries\source\, а затем выполните следующую команду:

```
emcc validate_product.cpp -s SIDE_MODULE=2 -O1
➥ -o validate_product.wasm
```

В окне консоли не должно отображаться никаких ошибок, а в исходной папке должен появиться новый файл `validate_product.wasm`.

Затем нужно определить, какие стандартные функции библиотеки C используются в коде. В папке Chapter 8\8.1 EmDynamicLibraries\source\ откройте файл `validate_product.cpp` и закомментируйте операторы `include` для файлов `cstdlib` и `cstring`. Сохраните файл, но не закрывайте его, поскольку вам нужно будет тут же раскомментировать эти строки кода.

В командной строке выполните следующую команду, аналогичную той, которую вы запускали только что:

```
emcc validate_product.cpp -s SIDE_MODULE=2 -O1
➥ -o validate_product.wasm
```

На этот раз вы должны увидеть сообщение об ошибке, отображаемое в окне консоли, как на рис. 8.9. Сообщение указывает, что функция `strlen` не определена

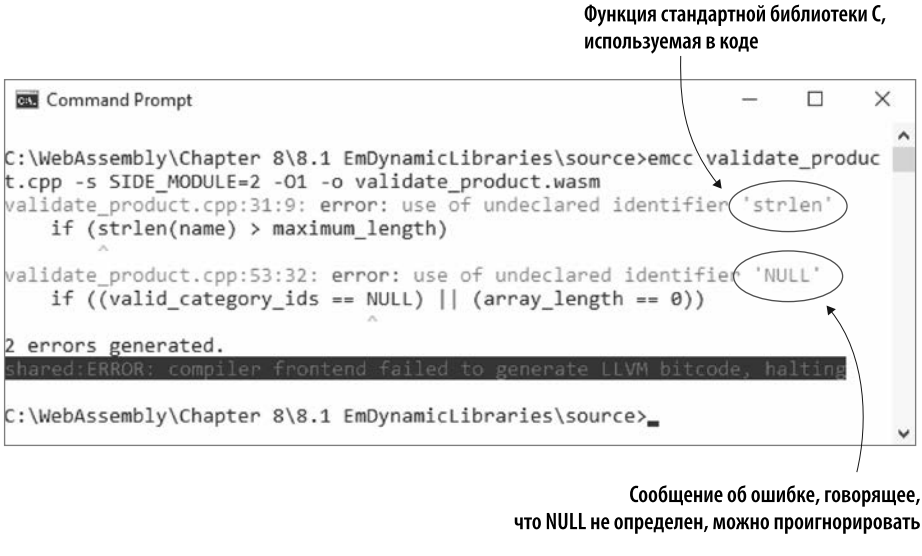


Рис. 8.9. Emscripten отображает сообщения о том, что функция strlen и NULL не определены

и что `NULL` не определен, но последнюю ошибку можно проигнорировать, поскольку не нужно ничего делать, чтобы добавить `NULL`. Обратите внимание на функцию `strlen`, так как нужно будет включить ее при использовании Emscripten для генерации основного модуля.

В файле `validate_product.cpp` удалите комментарии перед заголовочными файлами `cstdlib` и `cstring`. Затем сохраните файл.

После того как модуль WebAssembly для страницы Edit Product (Изменить товар) будет создан, нужно создать модуль формы Place Order (Разместить заказ). Как показано на рис. 8.10, вы будете следовать тому же процессу.

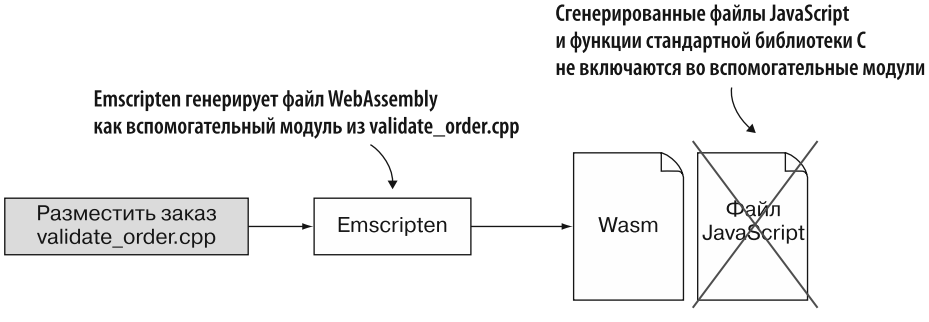


Рис. 8.10. Emscripten используется для генерации модуля WebAssembly, выполняющего проверку формы Place Order (Разместить заказ)

Генерация вспомогательного модуля для размещения заказа: validate_order.cpp

Как и в случае с модулем для страницы Edit Product (Изменить товар), прежде чем вы попытаетесь определить, какие стандартные функции библиотеки C использует этот модуль, необходимо убедиться, что код компилируется без проблем. Откройте командную строку, перейдите в папку Chapter 8\8.1 EmDynamicLibraries\source\, а затем выполните следующую команду:

```
emcc validate_order.cpp -s SIDE_MODULE=2 -O1
➥ -o validate_order.wasm
```

В окне консоли не должно отображаться никаких ошибок, а в исходной папке должен появиться новый файл `validate_order.wasm`.

Чтобы определить, использует ли код какие-либо функции стандартной библиотеки C, откройте файл `validate_order.cpp` и закомментируйте оператор `include` для

заголовочного файла `cstdlib`. Сохраните файл, но не закрывайте его, поскольку вам нужно будет тут же раскомментировать эту строку кода.

В командной строке выполните команду, аналогичную той, что вы запускали до этого:

```
emcc validate_order.cpp -s SIDE_MODULE=2 -O1
➥ -o validate_order.wasm
```

В окне консоли вы должны увидеть сообщение об ошибке, подобное тому, что показано на рис. 8.11. Оно указывает, что функция `atoi` не определена. Запомните эту функцию, поскольку нужно будет включить ее при использовании Emscripten для создания основного модуля. И снова можно спокойно игнорировать ошибку о том, что `NULL` есть необъявленный идентификатор.

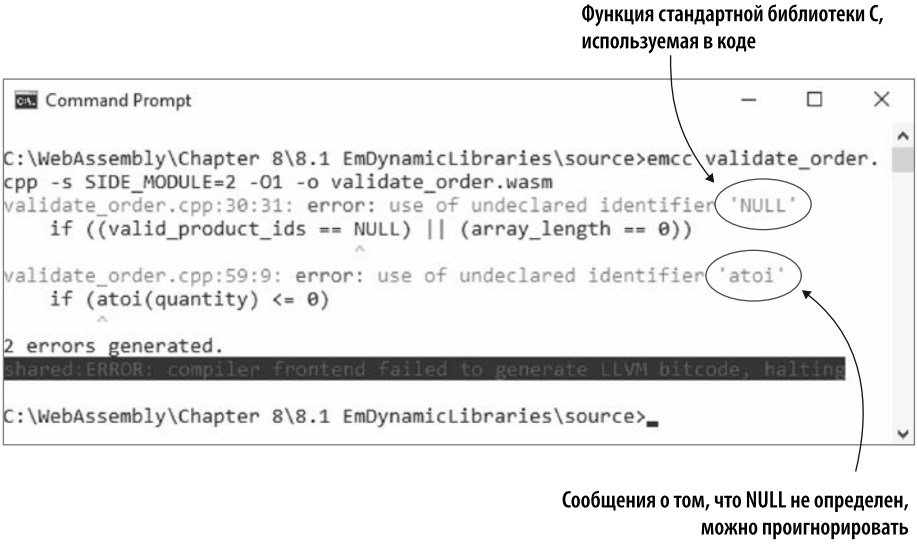


Рис. 8.11. Emscripten показывает сообщения о том, что функция atoi и NULL не определены

В файле `validate_order.cpp` удалите комментарий перед заголовочным файлом `cstdlib`. Затем сохраните файл.

После того как оба вспомогательных модуля будут созданы, можно перейти к созданию JavaScript-кода, который будет использоваться основным модулем (рис. 8.12).

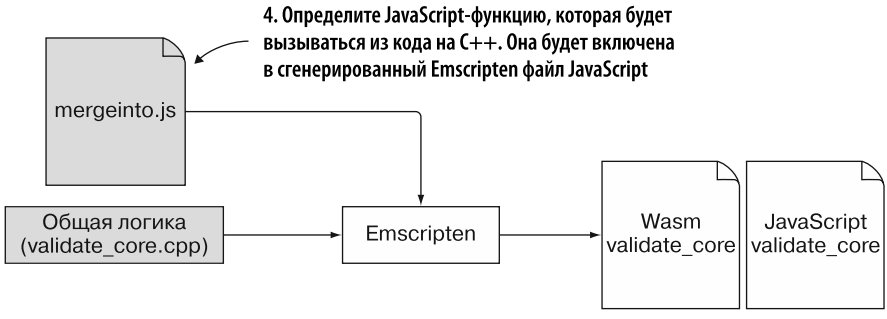


Рис. 8.12. Определите JavaScript-функцию, которая будет вызываться из кода на C++ в случае ошибки валидации. Код в этом файле будет включен в сгенерированный Emscripten файл JavaScript

8.1.4. Определение JavaScript-функции для обработки ошибок при валидации

В главе 5 мы создали файл `mergeinto.js` с JavaScript-функцией `UpdateHostAboutError`, которую функции на C++ будут вызывать, если возникнет проблема с валидацией. Функция `UpdateHostAboutError` прочитает сообщение из памяти модуля, а затем передаст строку в основной код JavaScript веб-страницы.

Как показано в следующем фрагменте кода, функция `UpdateHostAboutError` является частью объекта JavaScript, переданного как второй параметр функции `mergeInto`. Последняя добавит вашу функцию в объект `LibraryManager.library` в Emscripten, который добавляется в созданный Emscripten файл JavaScript:

```
mergeInto(LibraryManager.library, {
  UpdateHostAboutError: function(errorMessagePointer) {
    setErrorMessage(Module.UTF8ToString(errorMessagePointer));
  }
});
```

Скопируйте файл `mergeinto.js` из папки Chapter 5\5.1.1 EmJsLibrary\source\ в папку Chapter 8\8.1 EmDynamicLibraries\source\. Если используете Emscripten для создания модуля WebAssembly на следующем шаге, то также дадите ему указание добавить JavaScript из файла `mergeinto.js` в автоматически сгенерированный файл JavaScript. Для этого необходимо указать файл `mergeinto.js` с помощью параметра командной строки `--js-library`.

После того как файл `mergeinfo.js` будет получен, можно сделать следующий шаг (рис. 8.13) и сгенерировать основной модуль WebAssembly.

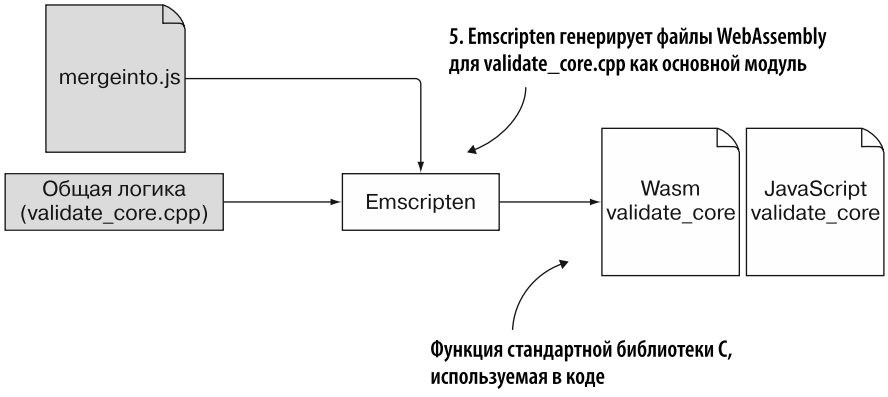


Рис. 8.13. Используйте Emscripten для генерации основного модуля WebAssembly из файла validate_core.cpp. Дайте Emscripten указание включить содержимое файла mergeinto.js в сгенерированный файл JavaScript

8.1.5. Использование Emscripten для генерации основного модуля WebAssembly

Чтобы Emscripten сгенерировал основной модуль, необходимо добавить параметр `MAIN_MODULE`. Если указать значение `1` этого параметра (`-s MAIN_MODULE=1`), то Emscripten отключит удаление *неиспользуемого кода*.

СПРАВКА

Удаление неиспользуемого кода предотвращает включение функций, которые не применяются кодом, в получившийся модуль WebAssembly.

Желательно отключать удаление неиспользуемого кода для основного модуля, так как он не знает, что понадобится вспомогательным модулям. В результате он сохраняет все функции, определенные в коде, и все функции стандартной библиотеки C. Для большого приложения желательно применять этот подход, поскольку код, вероятно, будет задействовать довольно много функций стандартной библиотеки C.

Если в коде используется лишь небольшое количество функций стандартной библиотеки C, как в данном случае, то все эти добавляемые дополнительные функции просто увеличивают размер модуля и замедляют загрузку и создание экземпляра. В таком случае нужно включить удаление неиспользуемого кода для основного модуля; для этого укажите `2` как значение `MAIN_MODULE`:

```
-s MAIN_MODULE=2
```

ПРЕДУПРЕЖДЕНИЕ

Прежде чем включить удаление неиспользуемого кода для основного модуля, убедитесь, что необходимые функции вспомогательных модулей остаются активными.

Создавая модули WebAssembly `validate_product` и `validate_order`, вы определили, что им нужны следующие функции стандартной библиотеки C: `strlen` и `atoi`. Чтобы дать Emscripten указание включить эти функции в сгенерированный модуль, добавьте их в массив командной строки `EXPORTED_FUNCTIONS`.

Код на JavaScript будет использовать вспомогательные функции `ccall`, `stringToUTF8` и `UTF8ToString` в Emscripten, поэтому нужно будет включить их в созданный JavaScript-файл. Для этого добавьте их в массив `EXTRA_EXPORTED_RUNTIME_METHODS` в командной строке при запуске компилятора Emscripten.

Обычно при создании модуля WebAssembly сгенерированный Emscripten код JavaScript существует как глобальный объект `Module`. Это работает, когда на каждой веб-странице есть только один модуль WebAssembly, но для сценария в данной главе мы создадим второй экземпляр модуля WebAssembly:

- один экземпляр для формы Edit Product (Изменить товар);
- один экземпляр для формы Place Order (Разместить заказ).

Чтобы это заработало, укажите параметр командной строки `-s MODULARIZE=1` — он приведет к тому, что объект `Module` в созданном Emscripten коде JavaScript будет заключен в функцию.

СПРАВКА

Если вы не используете параметр MODULARIZE, то простое добавление ссылки на файл JavaScript в Emscripten на веб-странице приведет к загрузке и созданию экземпляра модуля WebAssembly в момент загрузки файла страницей. Однако в случае использования параметра MODULARIZE вы сами несете ответственность за создание экземпляра объекта Module в JavaScript-коде для запуска загрузки и создания экземпляра модуля WebAssembly.

Откройте командную строку, перейдите в папку Chapter 8\8.1 EmDynamicLibraries\source\ и выполните следующую команду, чтобы создать модуль WebAssembly `validate_core`:

```
emcc validate_core.cpp --js-library mergeinto.js -s MAIN_MODULE=2
➥ -s MODULARIZE=1
➥ -s EXPORTED_FUNCTIONS=['_strlen','_atoi']
➥ -s EXTRA_EXPORTED_RUNTIME_METHODS=['ccall','stringToUTF8',
➥'UTF8ToString'] -o validate_core.js
```

После того как модули WebAssembly будут созданы, можно сделать следующие шаги (рис. 8.14) — скопировать файлы WebAssembly и сгенерированный Emscripten файл JavaScript туда, где они будут использоваться сайтом. Изменим HTML-код веб-страницы, чтобы в нем появился раздел формы Place Order (Разместить заказ). Затем обновим JavaScript-код, чтобы реализовать динамическое связывание модулей.

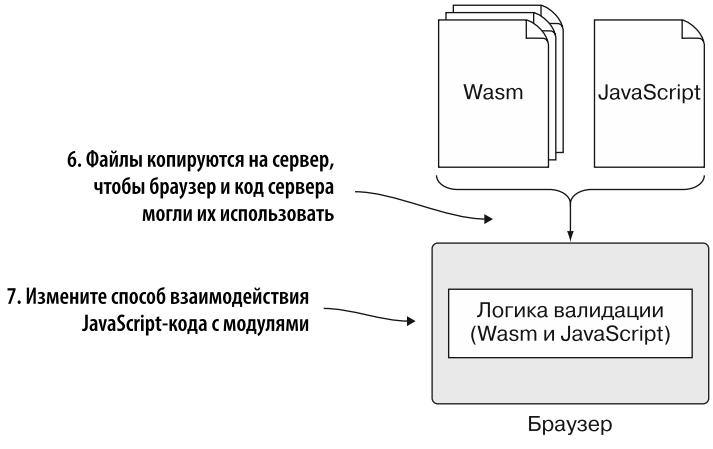


Рис. 8.14. Изменим HTML-код, чтобы в нем появилась форма Place Order (Разместить заказ), и обновим JavaScript-код, чтобы реализовать динамическое связывание модулей WebAssembly в браузере

8.2. НАСТРОЙКА ВЕБ-СТРАНИЦЫ

В папке Chapter 8\8.1 EmDynamicLibraries\ создайте папку frontend\, а затем скопируйте в нее следующие файлы:

- `validate_core.js`, `validate_core.wasm`, `validate_product.wasm` и `validate_order.wasm` из папки Chapter 8\8.1 EmDynamicLibraries\source\folder;
- `editproduct.html` и `editproduct.js` из папки Chapter 5\5.1.1 EmJsLibrary\frontend\.

Поскольку форма Place Order (Разместить заказ) будет добавлена на ту же веб-страницу, что и форма Edit Product (Изменить товар), следует переименовать файлы, чтобы названия были более общими. Переименуйте `editproduct.html` в `index.html` и `editproduct.js` в `index.js`.

Откройте файл `index.html` с помощью вашего любимого редактора, чтобы добавить новую панель навигации и элементы управления для формы Place Order

(Разместить заказ), как показано на рис. 8.15. Чтобы создать на веб-странице раздел навигации для таких вещей, как меню, будем использовать тег `Nav`.

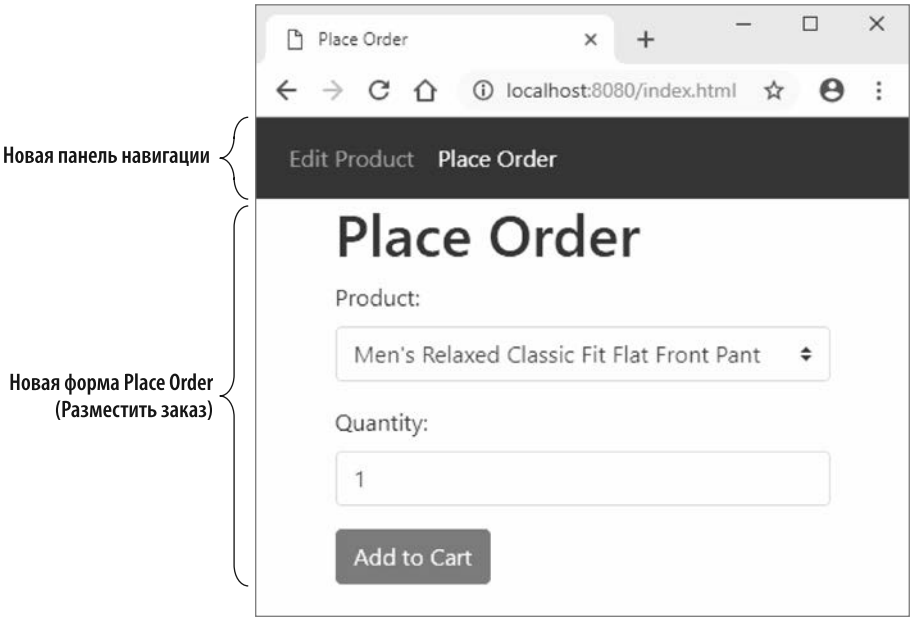


Рис. 8.15. Новая панель навигации и форма Place Order (Разместить заказ) контролируют данные, добавляемые на веб-страницу

При создании систем меню обычной практикой является определение элементов меню с помощью тегов `UL` и `LI`, а затем использование CSS для их стилизации. Тег `UL` расшифровывается как *Unordered List* (неупорядоченный список), в котором применяются маркеры. Тег `OL`, обозначающий *Ordered List* (упорядоченный или нумерованный список), также может использоваться, но это менее распространенный подход. В теге `UL` вы указываете один или несколько тегов `LI` (list item, элемент списка) для каждого пункта меню. Дополнительную информацию о создании панелей навигации можно найти на сайте www.w3schools.com/css/css_navbar.asp.

Между тегом `<body onload="initializePage()">` и первым открывающим тегом `div` (`<div class="container">`) в файле `index.html` добавьте HTML-код для новой панели навигации, показанный в листинге 8.6.

Добавьте атрибут `id` к тегу `H1` с названием `formTitle`, чтобы JavaScript-код мог изменять отображаемое пользователю значение, указывающее на то, какая форма открыта. Удалите текст из тега. Тег должен выглядеть так:

```
<h1 id="formTitle"></h1>
```

Листинг 8.6. HTML-код для новой панели навигации

```
...
<nav class="navbar navbar-expand-sm bg-dark navbar-dark">
  <ul class="navbar-nav">
    <li class="nav-item">
      <a id="navEditProduct" class="nav-link" href="#Edit Product"
          onclick="switchForm(true)">Edit Product</a>
    </li>
    <li class="nav-item">
      <a id="navPlaceOrder" class="nav-link" href="#PlaceOrder"
          onclick="switchForm(false)">Place Order</a>
    </li>
  </ul>
</nav>
...
```

Новая панель навигации

Щелчок на ссылке будет открывать форму Edit Product (Изменить товар)

Щелчок на этой ссылке будет открывать форму Place Order (Разместить заказ)

Поскольку элементы управления формы Edit Product (Изменить товар) нужно будет скрыть при отображении формы Place Order (Разместить заказ), заключите их в блок `div`, который можно будет показать или скрыть кодом на JavaScript. Добавьте открывающий тег `div` со значением `id`, равным `productForm`, перед тегом `div`, окружающим поле `Name`. Поскольку при первой загрузке веб-страницы может отображаться форма Place Order (Разместить заказ), а не Edit Product (Изменить товар), то следует добавить также атрибут стиля в тег `div productForm`, чтобы он был скрыт по умолчанию. Добавьте закрывающий тег `div` после тега кнопки `save`.

Измените значение `onclick` кнопки Save (Сохранить) с `onClickSave` на `onClick-SaveProduct`, чтобы было очевидно, что функция сохранения предназначена для формы Edit Product (Изменить товар). HTML-код для элементов управления этой формы в `index.html` должен соответствовать HTML-коду, показанному в листинге 8.7.

Листинг 8.7. Измененный HTML-код для раздела с формой Edit Product (Изменить товар) в index.html

```
...
<div id="productForm" style="display:none;">
  <div class="form-group">
    <label for="name">Name:</label>
    <input type="text" class="form-control" id="name">
  </div>
  <div class="form-group">
    <label for="category">Category:</label>
    <select class="custom-select" id="category">
      <option value="0"></option>
      <option value="100">Jeans</option>
      <option value="101">Dress Pants</option>
    </select>
```

Новый открывающий тег div окружает элементы управления формы Edit Product (Изменить товар)

```
  </div>

  <button type="button" class="btn btn-primary"
      onclick="onClickSaveProduct()">Save</button>
</div>
...
```

Значение onclick изменено на onClickSaveProduct

Был добавлен закрывающий тег div для тега productForm

Теперь нужно добавить элементы управления формы Place Order (Разместить заказ) в HTML. Как и в случае с элементами управления формы Edit Product (Изменить товар), обернем элементы управления формы Place Order (Разместить заказ) блоком `div` со значением `id`, равным `orderForm`.

Форма Place Order (Разместить заказ) будет иметь три элемента управления:

- раскрывающийся список продуктов;
- текстовое поле Quantity (Количество);
- кнопку Add to Cart (Добавить в корзину).

Добавьте HTML-код, показанный в листинге 8.8, после закрывающего тега `div`, добавленного для тега `div productForm` в файле `index.html`.

Листинг 8.8. Новый HTML-код для формы Place Order (Разместить заказ)

```
...

<div id="orderForm" style="display:none;">
  <div class="form-group">
    <label for="product">Product:</label>
    <select class="custom-select" id="product">
      <option value="0"></option>
      <option value="200">Women's Mid Rise Skinny Jeans</option>
      <option value="301">
        Men's Relaxed Classic Fit Flat Front Pant
      </option>
    </select>
  </div>
  <div class="form-group">
    <label for="quantity">Quantity:</label>
    <input type="text" class="form-control" id="quantity" value="0">
  </div>

  <button type="button" class="btn btn-primary"
      onclick="onClickAddToCart()">Add to Cart</button>
</div>
...
```

Последние изменения, которые нужно будет внести, — ссылки на файлы JavaScript в конце файла `index.html`:

- поскольку вы переименовали файл `editproduct.js` в `index.js`, то измените значение атрибута `src` первого тега `script` на `index.js`;
- когда вы использовали Emscripten для создания основного модуля, вы дали ему имя `validate_core.js`, поэтому нужно будет изменить значение атрибута `src` второго тега `script` на `validate_core.js`.

Два тега `script` должны выглядеть так:

```
<script src="index.js"></script>
<script src="validate_core.js"></script>
```

Мы изменили HTML, добавив новую панель навигации и элементы управления новой формы заказа. Теперь пришло время изменить JavaScript для работы с новыми модулями WebAssembly.

8.2.1. Изменение кода JavaScript веб-страницы

Откройте файл `index.js` в своем любимом редакторе. Этот файл теперь будет обрабатывать логику для двух форм: Edit Product (Изменить товар) и Place Order (Разместить заказ). Поэтому первое, что вам нужно сделать, — изменить имя объекта `initialData`, чтобы было ясно, что этот объект предназначен для формы Edit Product (Изменить товар). Измените имя с `initialData` на `initialProductData`, как показано ниже:

```
const initialProductData = {
  name: "Women's Mid Rise Skinny Jeans",
  categoryId: "100",
};
```

Раскрывающийся список товаров в форме Place Order (Разместить заказ) необходимо будет проверить, чтобы убедиться, что пользователь выбрал правильный идентификатор. Для этого передадим в модуль WebAssembly формы Place Order (Разместить заказ) массив, определяющий допустимые идентификаторы. Добавьте следующий глобальный массив допустимых идентификаторов после массива `VALID_CATEGORY_IDS` в файле `index.js`:

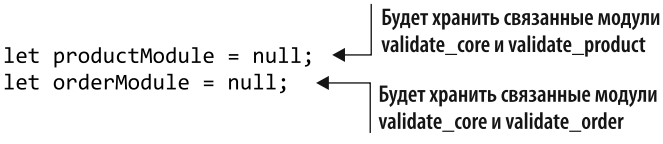
```
const VALID_PRODUCT_IDS = [200, 301];
```

Скомпилировав основной модуль (`validate_core.wasm`), вы дали Emscripten инструкцию обернуть его объект `Module` в функцию, чтобы можно было создать несколько экземпляров данного объекта. Это нужно сделать, так как для этой веб-страницы будут созданы два экземпляра модуля WebAssembly.

Форма Edit Product (Изменить товар) будет использовать экземпляр WebAssembly, в котором основной модуль связан с вспомогательным модулем изменения

товара: `validate_product.wasm`. Форма Place Order (Разместить заказ) также будет использовать экземпляр WebAssembly, в котором основной модуль связан со вспомогательным модулем этой формы : `validate_order.wasm`.

Чтобы хранить эти два экземпляра модуля Emscripten, необходимо добавить глобальные переменные в следующем фрагменте кода после массива `VALID_PRODUCT_IDS` в файле `index.js`:



Вот и все изменения для глобальных объектов. Теперь нужно внести несколько изменений в функцию `initializePage`.

Функция initializePage

Первое, что необходимо изменить для функции `initializePage`, — имя объекта, используемого для заполнения поля имени и раскрывающегося списка категорий. Имя объекта следует изменить с `initialData` на `initialProductData`.

Данная веб-страница создается как SPA, поэтому щелчок на ссылке на панели навигации не приведет к переходу на новую страницу. Вместо этого *идентификатор фрагмента* помещается в конец адреса в адресной строке браузера, а содержимое веб-страницы корректируется, отображая желаемое представление. Если бы вы дали кому-то адрес веб-страницы с идентификатором фрагмента, то она отобразила бы этот раздел, словно пользователь перешел к нему, щелкнув на ссылке навигации.

СПРАВКА

Идентификатор фрагмента — это необязательная часть в конце URL, которая начинается с символа решетки (#), как показано на рис. 8.16. Обычно он используется для обозначения раздела веб-страницы. Когда вы щелкаете на гиперссылке, указывающей на идентификатор фрагмента, веб-страница открывается в этом месте, что полезно при навигации по большим документам.

Поскольку нам нужно, чтобы веб-страница отображала правильное представление в зависимости от того, был ли в адресе страницы указан идентификатор фрагмента, добавим код в конец функции `initializePage`, чтобы проверить, был ли включен идентификатор. По умолчанию на веб-странице отображается форма Edit Product (Изменить товар); но, если в адрес включен идентификатор #PlaceOrder, вместо этого вы увидите форму Place Order (Разместить заказ). После кода обнаружения идентификатора фрагмента добавим вызов функции, которая приведет к отображению правильной формы.

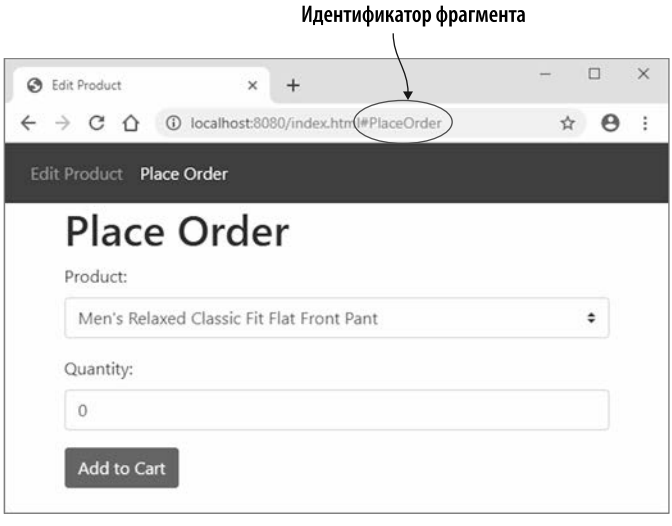


Рис. 8.16. URL веб-страницы с идентификатором фрагмента PlaceOrder

Измените функцию `initializePage` в файле `index.js`, чтобы она соответствовала коду, показанному в листинге 8.9.

Листинг 8.9. Измененная функция initializePage

```
...
function initializePage() {
  document.getElementById("name").value = initialProductData.name;

  const category = document.getElementById("category");
  const count = category.length;
  for (let index = 0; index < count; index++) {
    if (category[index].value === initialProductData.categoryId) {
      category.selectedIndex = index;
      break;
    }
  }

  let showEditProduct = true;
  if ((window.location.hash) &&
      (window.location.hash.toLowerCase() === "#placeorder")) {
    showEditProduct = false;
  }

  switchForm(showEditProduct);
}
...
```

initialData было изменено на initialProductData

initialData было изменено на initialProductData

По умолчанию отображает форму Edit Product (Изменить товар)

Если в адрес сайта добавлен идентификатор фрагмента и он равен #placeorder…

…будет отображена форма Place Order (Разместить заказ)

Отображает нужную форму

Нужно будет создать функцию `switchForm` для настройки веб-страницы таким образом, чтобы она отображала запрошенную форму: Edit Product (Изменить товар) или Place Order (Разместить заказ).

Функция switchForm

Функция `switchForm` выполнит следующие действия:

- удалит все сообщения об ошибках, которые могут отображаться;
- выделит элемент на панели навигации, соответствующий форме, которая должна отображаться;
- изменит заголовок в теге `H1` на веб-странице, чтобы показать название текущей формы;
- покажет запрошенную форму и скроет другую.

Основной модуль был скомпилирован с флагом `MODULARIZE`, поэтому Emscripten не загружает модуль WebAssembly автоматически и не создает его экземпляр. Вам решать, когда именно создать экземпляр объекта `Module` из Emscripten.

Если экземпляр этого объекта еще не был создан для запрошенной формы, то функция `switchForm` создаст и его. Объект `Module` в Emscripten может принимать объект JavaScript для управления выполнением кода, поэтому ваш код будет использовать его для передачи имени вспомогательного модуля, с которым он должен связываться через свойство массива `dynamicLibraries`.

Добавьте код, показанный в листинге 8.10, после функции `initializePage` в файле `index.js`.

Листинг 8.10. Функция switchForm

```
...

function switchForm(showEditProduct) {
  setErrorMessage("");
  setActiveNavLink(showEditProduct);
  setFormTitle(showEditProduct);

  if (showEditProduct) {
    if (productModule === null) {
      productModule = new Module({
        dynamicLibraries: ['validate_product.wasm']
      });
    }

    showElement("productForm", true);
    showElement("orderForm", false);
  } else {
```

Подсвечивает заданный элемент панели навигации

Должна отображаться форма Edit Product (Изменить товар)

Изменяет заголовок на нужный

Создает экземпляр только в том случае, если он не был создан ранее

Создает новый экземпляр основного модуля WebAssembly

Указывает Emscripten, что нужно связаться со вспомогательным модулем Product

Должна отображаться форма заказа

Показывает форму Edit Product (Изменить товар) и скрывает форму заказа

```
    if (orderModule === null) {
      orderModule = new Module({
      dynamicLibraries: ['validate_order.wasm']
      });
    }

    showElement("productForm", false);
    showElement("orderForm", true);
  }
}
...
```

Создает новый экземпляр основного модуля WebAssembly

Указывает Emscripten, что нужно связаться со вспомогательным модулем Order

Скрывает форму Edit Product (Изменить товар) и показывает форму заказа

Следующая функция, которую нужно создать, — `setActiveNavLink`, которая выделит подсветкой часть панели навигации для отображаемой формы.

Функция setActiveNavLink

Поскольку для элементов панели навигации можно указать несколько имен классов CSS, будем использовать объект `classList` элемента DOM, который позволяет вставлять и удалять отдельные имена классов. Ваша функция будет следить за тем, чтобы из обоих элементов панели навигации имя класса `"active"` было удалено, а затем добавлено только к элементу панели навигации для отображаемого представления.

Добавьте функцию `setActiveNavLink`, показанную в листинге 8.11, после функции `switchForm` в файле `index.js`.

Листинг 8.11. Функция setActiveNavLink

```
...

function setActiveNavLink(Editproduct) {
  const navEditProduct = document.getElementById("navEditProduct");
  const navPlaceOrder = document.getElementById("navPlaceOrder");
  navEditProduct.classList.remove("active");
  navPlaceOrder.classList.remove("active");

  if (editProduct) { navEditProduct.classList.add("active"); }
  else { navPlaceOrder.classList.add("active"); }
}
...
```

Удаляет имя класса "active" из обоих элементов

Добавляет имя класса "active" к элементу, соответствующему отображаемой форме

Следующая функция, которую нужно создать, — `setFormTitle`, которая будет корректировать текст на веб-странице, чтобы указать, какая форма отображается.

Функция setFormTitle

После функции `setActiveNavLink` в файле `index.js` добавьте функцию `setFormTitle` для отображения заголовка формы в теге `H1` на веб-странице:

```
function setFormTitle(editProduct) {
  const title = (editProduct ? "Edit Product" : "Place Order");
  document.getElementById("formTitle").innerText = title;
}
```

Изначально нужно было отображать или скрывать только раздел ошибок веб-страницы, поэтому код для отображения или сокрытия элемента был частью функции `setErrorMessage`. Теперь, когда мы добавили дополнительные элементы веб-страницы, которые необходимо показать или скрыть, переместим эту логику в отдельную функцию.

Функция showElement

Добавьте функцию `showElement` после функции `setFormTitle` в файл `index.js`, как показано ниже:

```
function showElement(elementId, show) {
  const element = document.getElementById(elementId);
  element.style.display = (show ? "" : "none");
}
```

Для проверки формы заказа потребуется получить идентификатор выбранного пользователем товара из раскрывающегося списка. Функция `getSelectedCategoryId` уже получает выбранный пользователем идентификатор из раскрывающегося списка, но используется для раскрывающегося списка категорий формы Edit Product (Изменить товар). Теперь мы изменим эту функцию, сделав ее более общей, чтобы ее можно было использовать и в форме Place Order (Разместить заказ).

Функция getSelectedCategoryId

Измените имя функции `getSelectedCategoryId` на `getSelectedDropdownId` и добавьте `elementId` в качестве параметра. Внутри функции измените имя переменной с `category` на `dropdown` и замените строку `"category"` на `elementId` в вызове `getElementById`.

Функция `getSelectedDropdownId` должна соответствовать коду, показанному ниже:

```
function getSelectedDropdownId(elementId) {
  const dropdown = document.getElementById(elementId);
  const index = dropdown.selectedIndex;
  if (index !== -1) { return dropdown[index].value; }

  return "0";
}
```

Мы изменили имя функции и добавили параметр elementId

Имя переменной изменилось, и elementId теперь передается в getElementById

После того как функция `showElement` для отображения или сокрытия элементов на веб-странице будет создана, можно изменить функцию `setErrorMessage`, чтобы вызывать новую функцию, а не напрямую настраивать видимость элемента.

Функция setErrorMessage

Измените функцию `setErrorMessage` в файле `index.js` так, чтобы она вызывала функцию `showElement`, а не напрямую задавала стиль элемента. Функция должна выглядеть следующим образом:

```
function setErrorMessage(error) {
  const errorMessage = document.getElementById("errorMessage");
  errorMessage.innerText = error;
  showElement("errorMessage", (error !== ""));
}
```

Показывает элемент errorMessage в случае ошибки и скрывает его, если ошибки нет

Поскольку на веб-странице теперь будет два набора элементов управления, наличие функции `onClickSave` может сбивать с толку, поэтому переименуем ее так, чтобы имя указывало на то, что функция используется формой Edit Product (Изменить товар).

Функция onClickSave

Переименуйте функцию `onClickSave` в `onClickSaveProduct`. Поскольку вы переименовали функцию `getSelectedCategoryId` в `getSelectedDropdownId`, необходимо переименовать вызов функции. Также передайте идентификатор раскрывающегося списка (`"category"`) как параметр функции `getSelectedDropdownId`.

Функция `onClickSaveProduct` должна соответствовать коду, показанному в листинге 8.12.

Листинг 8.12. Функция onClickSave, переименованная в onClickSaveProduct

```
...
function onClickSaveProduct() {
  setErrorMessage("");

  const name = document.getElementById("name").value;
  const categoryId = getSelectedDropdownId("category");

  if (validateName(name) && validateCategory(categoryId)) {

  }
}
...
```

Мы изменили имя — было onClickSave

Изменили имя функции и определили идентификатор раскрывающегося списка

Проблем не обнаружено. Данные могут быть переданы в код серверной стороны

Поскольку основной модуль был скомпилирован с флагом `MODULARIZE`, нужно было создать экземпляр объекта `Module` в Emscripten. Функции `validateName` и `validateCategory` необходимо изменить так, чтобы они вызывали созданный экземпляр `Module`, `productModule`, а не объект `Module` в Emscripten.

Функции validateName и validateCategory

Нужно изменить все места в функциях `validateName` и `validateCategory`, которые вызывают объект `Module` в Emscripten, чтобы вместо этого использовать экземпляр `Module` — `productModule`. Функции `validateName` и `validateCategory` в `index.js` должны соответствовать коду, показанному в листинге 8.13.

Листинг 8.13. Измененные функции validateName и validateCategory

```
...

function validateName(name) {
  const isValid = productModule.ccall('ValidateName',
      'number',
      ['string', 'number'],
      [name, MAXIMUM_NAME_LENGTH]);

  return (isValid === 1);
}

function validateCategory(categoryId) {
  const arrayLength = VALID_CATEGORY_IDS.length;
  const bytesPerElement = productModule.HEAP32.BYTES_PER_ELEMENT;
  const arrayPointer = productModule._malloc((arrayLength *
      bytesPerElement));
  productModule.HEAP32.set(VALID_CATEGORY_IDS,
        (arrayPointer / bytesPerElement));

  const isValid = productModule.ccall('ValidateCategory',
      'number',
      ['string', 'number', 'number'],
      [categoryId, arrayPointer, arrayLength]);

  productModule._free(arrayPointer);

  return (isValid === 1);
}
```

Модуль был заменен на productModule

Модуль был заменен на productModule

Изменив существующий код изменения товара, можно перейти к добавлению кода размещения заказа. Первый шаг — создание функции `onClickAddToCart`.

Функция onClickAddToCart

Функция `onClickAddToCart` для формы Place Order (Разместить заказ) будет очень похожа на функцию `onClickSaveProduct` формы Edit Product (Изменить товар).

Здесь мы получаем выбранный идентификатор из раскрывающегося списка товаров, а также введенное пользователем значение количества. Затем вызовем функции `validateProduct` и `validateQuantity` в JavaScript для вызова модуля WebAssembly и проверки введенных пользователем значений. Если нет проблем с валидацией, то данные могут быть сохранены.

Добавьте код, показанный в листинге 8.14, после функции `validateCategory` в файле `index.js`.

Листинг 8.14. Функция onClickAddToCart в файле index.js

```
...

function onClickAddToCart() {
  setErrorMessage("");

  const productId = getSelectedDropdownId("product");
  const quantity = document.getElementById("quantity").value;

  if (validateProduct(productId) &&
    validateQuantity(quantity)) {

  }
}
```

Получает выбранный пользователем идентификатор из раскрывающегося списка товаров

Получает введенное пользователем количество

Проверяет идентификатор товара

Проверяет количество

Проблем с введенными пользователем данными не обнаружено. Данные могут быть сохранены

Теперь нужно создать функцию `validateProduct`, которая будет вызывать модуль WebAssembly, чтобы проверять, действителен ли выбранный пользователем идентификатор товара.

Функция validateProduct

Функция `validateProduct` вызовет функцию `ValidateProduct` модуля, а она имеет следующую сигнатуру в C++:

```
int ValidateProduct(char* product_id,
int* valid_product_ids,
int array_length);
```

Функция `validateProduct` в JavaScript передаст функции модуля следующие параметры:

- выбранный пользователем идентификатор товара;
- массив допустимых идентификаторов;
- длину массива.

Передадим выбранный пользователем идентификатор товара в модуль в виде строки, а функция `ccall` в Emscripten обработает управление памятью строки самостоятельно, поскольку тип параметра указан как `'string'`.

Массив допустимых идентификаторов — это целые числа (32-битные), но функция `ccall` в Emscripten может управлять памятью массива самостоятельно, только если вы имеете дело с 8-битными целыми числами. В результате придется вручную выделить часть памяти модуля для хранения значений массива, а затем скопировать значения в память.

Передадим указатель места в памяти для действительных идентификаторов в функцию `ValidateProduct`. В WebAssembly указатели представлены как 32-битные целые числа, поэтому укажем данный тип параметра как `'number'`.

Добавьте функцию `validateProduct`, показанную в листинге 8.15, в конец файла `index.js`.

Листинг 8.15. Функция validateProduct в файле index.js

```
...

function validateProduct(productId) {
  const arrayLength = VALID_PRODUCT_IDS.length;
  const bytesPerElement = orderModule.HEAP32.BYTES_PER_ELEMENT;
  const arrayPointer = orderModule._malloc((arrayLength *
      bytesPerElement));
  orderModule.HEAP32.set(VALID_PRODUCT_IDS,
      (arrayPointer / bytesPerElement));

  const isValid = orderModule.ccall('ValidateProduct',
      'number',
      ['string', 'number', 'number'],
      [productId, arrayPointer, arrayLength]);

  orderModule._free(arrayPointer);

  return (isValid === 1);
}
```

Выделяет объем памяти, достаточный для всех элементов массива

Копирует элементы массива в память модуля

Вызывает функцию ValidateProduct в модуле

Освобождает память, выделенную для массива

Последняя необходимая функция в JavaScript — `validateQuantity`, которая будет вызывать модуль для проверки введенного пользователем количества.

Функция validateQuantity

Функция `validateQuantity` вызовет функцию `ValidateQuantity` модуля, которая имеет следующую сигнатуру в C++:

```
int ValidateQuantity(char* quantity);
```

Передадим введенное пользователем значение количества в модуль в виде строки, а функция `ccall` в Emscripten обработает управление памятью строки самостоятельно, поскольку тип параметра указан как `'string'`.

Добавьте функцию `validateQuantity`, показанную в следующем фрагменте кода, в конец файла `index.js`:

```
function validateQuantity(quantity) {
  const isValid = orderModule.ccall('ValidateQuantity',
      'number',
      ['string'],
      [quantity]);

  return (isValid === 1);
}
```

8.2.2. Просмотр результатов

Закончив изменять JavaScript-код, откройте браузер и введите `http://localhost:8080/index.html` в адресную строку, чтобы просмотреть веб-страницу. Вы можете проверить навигацию, щелкнув на ссылках на панели навигации. Как показано на рис. 8.17, отображаемое представление должно переключаться между формами Edit Product (Изменить товар) и Place Order (Разместить заказ), а в адресной строке должен быть указан идентификатор фрагмента в соответствии с последней нажатой ссылкой.

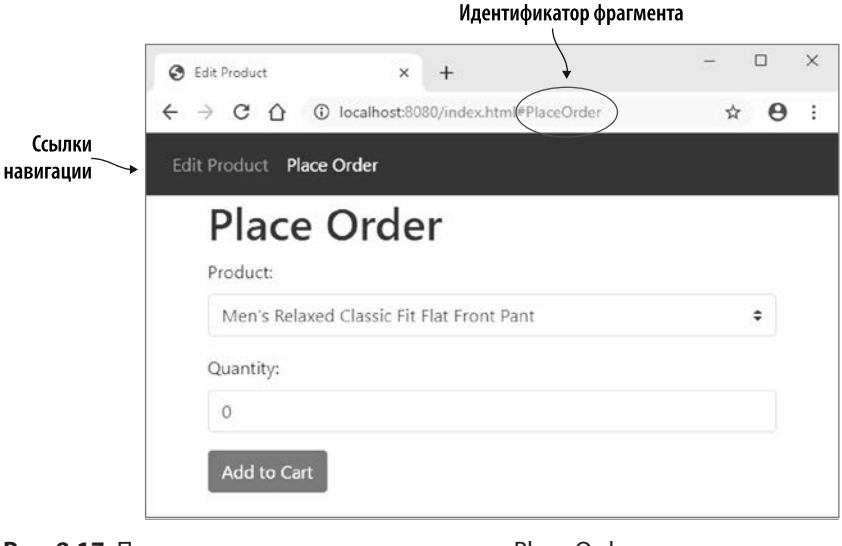


Рис. 8.17. При переходе по ссылке навигации Place Order показываются элементы управления формой Place Order (Разместить заказ), а идентификатор фрагмента добавляется в адресную строку браузера

Вы можете протестировать валидацию, выбрав элемент в раскрывающемся списке Product (Товар), оставив для параметра Quantity (Количество) значение 0 и нажав кнопку Add to Cart (Добавить в корзину). На веб-странице должна отображаться ошибка, как показано на рис. 8.18.

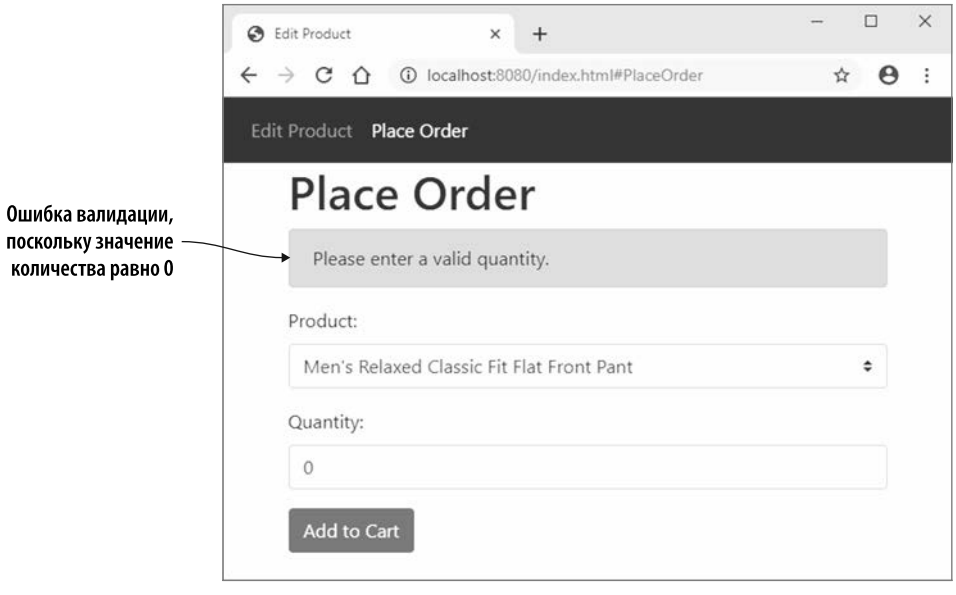


Рис. 8.18. Ошибка валидации в новой форме Place Order (Разместить заказ), когда заданное значение количества равно 0

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже приведены некоторые возможные варианты использования того, что вы узнали в этой главе.

- Если ваш модуль WebAssembly не нужно загружать и создавать экземпляр до некоторого момента после загрузки веб-страницы, то можете добавить флаг `-s MODULARIZE=1` при компиляции модуля. Это позволит вам контролировать, когда модуль будет загружен и запущен, что поможет ускорить время начальной загрузки сайта.
- Другой вариант применения флага `-s MODULARIZE=1` заключается в том, что он позволяет создавать несколько экземпляров модуля WebAssembly. Одно-

страничное приложение потенциально может работать долго, и можно уменьшить использование памяти, создав экземпляр модуля, когда это необходимо, и уничтожив его, если он больше не нужен (например, когда пользователь перешел к другой части приложения).

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Предположим, у вас есть вспомогательный модуль `process_fulfillment.wasm`. Как можно было бы создать новый экземпляр объекта `Module` в Emscripten и дать ему указание динамически связываться с этим вспомогательным модулем?
2. Какой флаг необходимо передать Emscripten при компиляции основного модуля WebAssembly, чтобы объект `Module` был заключен в функцию в созданном Emscripten файле JavaScript?

РЕЗЮМЕ

В этой главе вы узнали, как создать простое одностраничное приложение, использующее идентификатор фрагмента в URL, чтобы указать, какая форма должна отображаться.

Кроме того, вы узнали следующее.

- Можно создать несколько экземпляров JavaScript-объекта `Module` в Emscripten, добавив флаг `-s MODULARIZE=1` при компиляции основного модуля.
- Когда основной модуль компилируется с помощью флага `MODULARIZE`, настройки для объекта `Module` передаются в конструктор модуля как объект JavaScript.
- Удаление неиспользуемого кода может быть включено для основного модуля с помощью флага `-s MAIN_MODULE=2`. Однако тогда необходимо, чтобы вы явно указали, какие функции поддерживать для вспомогательных модулей, задействуя массив командной строки: `EXPORTED_FUNCTIONS`.
- Вы можете проверить, какие функции стандартной библиотеки C используются вспомогательным модулем, закомментировав заголовочные файлы и попытавшись скомпилировать модуль. Emscripten будет выдавать ошибки в командной строке, указывая, какие функции не определены.

9 Потоки: веб-воркеры и pthread

В этой главе

- ✓ Использование веб-воркеров для получения и компиляции модуля WebAssembly.
- ✓ Создание экземпляра модуля WebAssembly от имени JavaScript-кода в Emscripten.
- ✓ Создание модуля WebAssembly, использующего pthread.

В этой главе вы узнаете о различных вариантах использования *потоков* в браузере и их связи с модулями WebAssembly.

ОПРЕДЕЛЕНИЕ

Поток — это путь выполнения внутри процесса; процесс может иметь несколько потоков. Поток pthread, также известный как поток POSIX, представляет собой API, определенный стандартом POSIX.1c для исполняемого модуля, не зависящий от языка программирования (см. https://en.wikipedia.org/wiki/POSIX_Threads).

По умолчанию пользовательский интерфейс (user interface, UI) и JavaScript веб-страницы работают в одном потоке. Если код обрабатывает слишком много данных без периодического взаимодействия с UI, то интерфейс может перестать отвечать на запросы. Анимация зависнет, а элементы управления на веб-странице не будут реагировать на пользовательский ввод, что не очень хорошо.

Если веб-страница не отвечает достаточно долго (обычно около десяти секунд), то браузер может даже предложить пользователю прекратить выполнение сценария на странице, как показано на рис. 9.1. Если пользователь сделает это, то она может перестать работать должным образом, пока пользователь не обновит страницу.

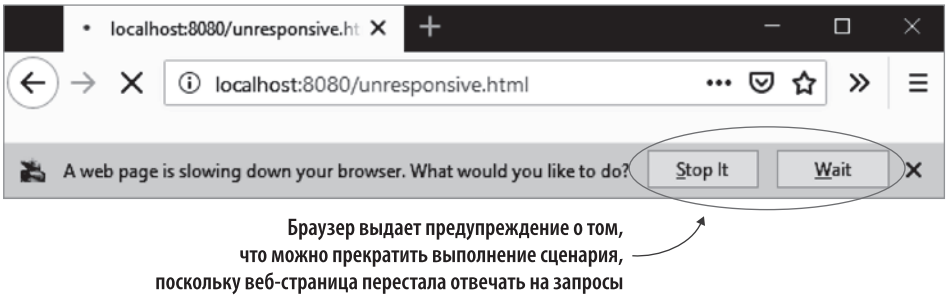


Рис. 9.1. Долгое выполнение процесса привело к тому, что Firefox перестал отвечать на запросы. Браузер выдает предупреждение о том, что можно прекратить выполнение сценария

СОВЕТ

Чтобы веб-страницы оставались максимально отзывчивыми, всякий раз при взаимодействии с веб-API, имеющим как синхронные, так и асинхронные функции, рекомендуется отдавать предпочтение асинхронным функциям.

Разработчикам браузеров нужна была возможность выполнять некую ресурсоемкую обработку, не вмешиваясь в пользовательский интерфейс, что привело к *созданию веб-воркеров*.

9.1. ПРЕИМУЩЕСТВА ВЕБ-ВОРКЕРОВ

Как работают веб-воркеры и почему их стоит использовать? Благодаря им можно создавать фоновые потоки в браузерах. Как показано на рис. 9.2, веб-воркеры позволяют запускать JavaScript в потоке, отдельном от потока пользовательского интерфейса; общение между ними осуществляется путем передачи сообщений.

В отличие от потока пользовательского интерфейса, при желании разрешается применять синхронные функции в веб-воркере, поскольку это не блокирует поток UI. Внутри воркера вы можете создавать дополнительные воркеры. Кроме того, вы получаете доступ ко многим из элементов, знакомым вам по потоку пользовательского интерфейса, — например, к fetch, WebSocket и IndexedDB. Полный список API, доступных для веб-воркеров, можно найти на странице MDN Web Docs: http://mng.bz/gVBG.

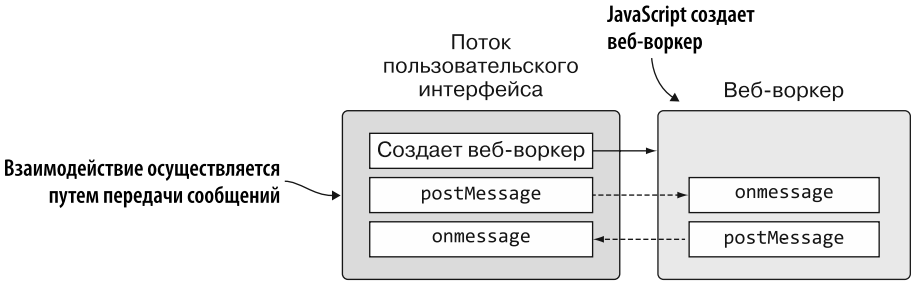


Рис. 9.2. JavaScript создает веб-воркер и взаимодействует с ним, передавая сообщения

Еще одним преимуществом веб-воркеров является то, что большинство современных устройств имеют несколько процессорных ядер. Если обработку можно разделить на несколько потоков, то время, необходимое для ее завершения, должно уменьшиться. Вдобавок веб-воркеры поддерживаются почти во всех браузерах, включая мобильные.

Модули WebAssembly могут использовать веб-воркеры несколькими способами.

- Как вы узнаете из раздела 9.3, веб-воркер можно применять для предварительной загрузки модуля WebAssembly. Веб-воркер может загрузить и скомпилировать модуль, а затем передать его в основной поток, который затем может создать экземпляр скомпилированного модуля и использовать его в обычном режиме.
- Emscripten поддерживает возможность создания двух модулей WebAssembly, один из которых находится в основном потоке, а другой — в веб-воркере. Эти два модуля будут взаимодействовать с помощью вспомогательных функций Emscripten, определенных в *Worker API* в Emscripten. В этой главе мы не будем рассматривать данный подход, но для многих функций Emscripten будут представлены JavaScript-версии. Чтобы узнать больше о Worker API в Emscripten, посетите страницу документации http://mng.bz/eD1q.

СПРАВКА

Вам нужно будет создать два файла на C или C++, чтобы скомпилировать один из них для работы в основном потоке, а другой — для работы в веб-воркере. Файл веб-воркера должен быть скомпилирован с параметром -s BUILD_AS_WORKER=1.

- Разрабатывается функция post-MVP, которая создает специальный вид веб-воркера, позволяющего модулю WebAssembly использовать pthread (потоки POSIX). На данный момент этот подход все еще считается экспериментальным, и в некоторых браузерах необходимо включить флаги, чтобы код мог работать. Вы узнаете об этом подходе в разделе 9.4, в котором я также более подробно расскажу о pthread.

9.2. РЕКОМЕНДАЦИИ ПО ИСПОЛЬЗОВАНИЮ ВЕБ-ВОРКЕРОВ

Вскоре вы научитесь использовать веб-воркеры, но прежде всего нужно учитывать следующее.

- У веб-воркеров высокая стоимость запуска и высокая стоимость памяти, поэтому они не предназначены для использования в большом количестве и ожидается, что время их жизни будет довольно долгим.
- Поскольку веб-воркеры работают в фоновом потоке, вы не имеете прямого доступа к функциям пользовательского интерфейса веб-страницы или DOM.
- Единственный способ общаться с веб-воркером — это отправлять вызовы `postMessage` и отвечать на сообщения через обработчик событий `onmessage`.
- Несмотря на то что обработка фонового потока не блокирует поток UI, вам все равно нужно помнить о ненужной обработке и использовании памяти, поскольку некоторые ресурсы устройства все еще будут применяться. В качестве аналогии, если пользователь задействует телефон, множество сетевых запросов может израсходовать тарифный план, а сложная обработка может израсходовать заряд батареи.
- В настоящий момент веб-воркеры доступны только в браузерах. Если ваш модуль WebAssembly, например, должен также поддерживать Node.js, то следует об этом помнить. Начиная с версии 10.5, Node.js включает экспериментальную поддержку *потоков воркеров*, но они еще не поддерживаются Emscripten. Больше информации о поддержке потоков воркеров в Node.js доступно на сайте https://nodejs.org/api/worker_threads.html.

9.3. ПРЕДВАРИТЕЛЬНАЯ ЗАГРУЗКА МОДУЛЯ WEBASSEMBLY С ПОМОЩЬЮ ВЕБ-ВОРКЕРА

Предположим, у вас есть веб-страница, которой потребуется модуль WebAssembly в какой-то момент после загрузки страницы. Вместо того чтобы загружать и создавать экземпляр модуля во время загрузки страницы, вы решаете отложить процесс до тех пор, пока страница не загрузится полностью, с целью максимально сократить время загрузки. Чтобы ваша веб-страница оставалась максимально отзывчивой, вы также решаете использовать веб-воркер для обработки загрузки и компиляции модуля WebAssembly в фоновом потоке.

Как показано на рис. 9.3, в этом разделе вы узнаете, как:

- создать веб-воркер;

- загрузить и скомпилировать модуль WebAssembly, пока он находится в веб-воркере;
- передавать и получать сообщения между основным потоком пользовательского интерфейса и воркером;
- изменить поведение Emscripten по умолчанию, в котором он обычно обрабатывает загрузку и создание экземпляра модуля WebAssembly и использует уже скомпилированный модуль.

Порядок предварительной загрузки модуля WebAssembly с помощью веб-воркера

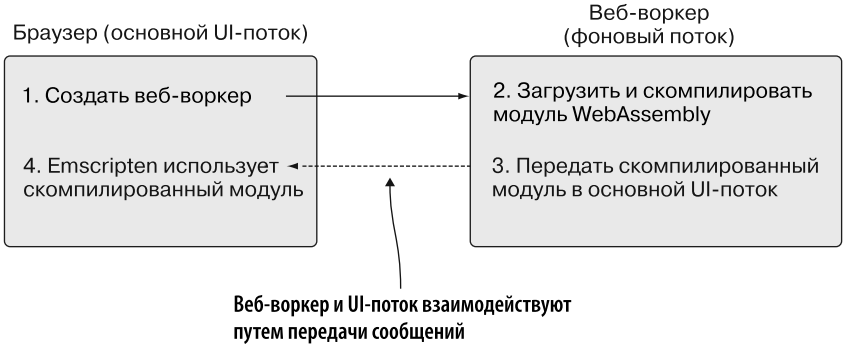


Рис. 9.3. Ваш JavaScript создает веб-воркер, который будет загружать и компилировать модуль WebAssembly, а затем передавать этот модуль в основной поток UI. Emscripten будет использовать этот скомпилированный модуль, вместо того чтобы самому его загружать

Следующие шаги определяют решение для этого сценария (рис. 9.4).

1. Измените логику `calculate_primes`, созданную в главе 7, чтобы определить, сколько времени потребуется для завершения вычислений.
2. Используйте Emscripten для создания файлов WebAssembly на основе логики `calculate_primes`.
3. Скопируйте сгенерированные файлы WebAssembly на сервер, чтобы браузер мог их использовать.
4. Создайте HTML и JavaScript для веб-страницы, на которой будет создан веб-воркер, и пусть JavaScript в Emscripten использует скомпилированный модуль WebAssembly, полученный от воркера.
5. Создайте файл JavaScript веб-воркера, который загрузит и скомпилирует модуль WebAssembly.

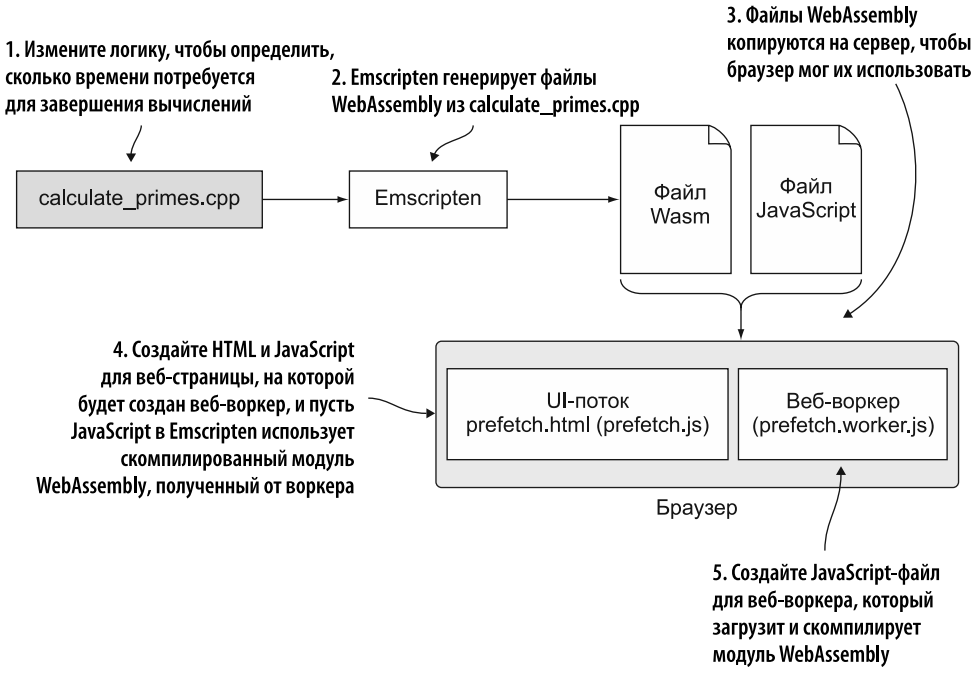


Рис. 9.4. Порядок реализации сценария предварительной загрузки. Измените файл calculate_primes.cpp, чтобы определить, сколько времени займут вычисления. Дайте Emscripten команду сгенерировать файлы WebAssembly, затем создайте HTML и JavaScript. JavaScript создаст веб-воркер для загрузки и компиляции модуля WebAssembly. Наконец, скомпилированный модуль будет передан обратно на веб-страницу, где его экземпляр будет создан вашим кодом вместо JavaScript в Emscripten

Первый шаг, показанный на рис. 9.5, — изменение логики `calculate_primes` с целью определить, сколько времени займут вычисления.

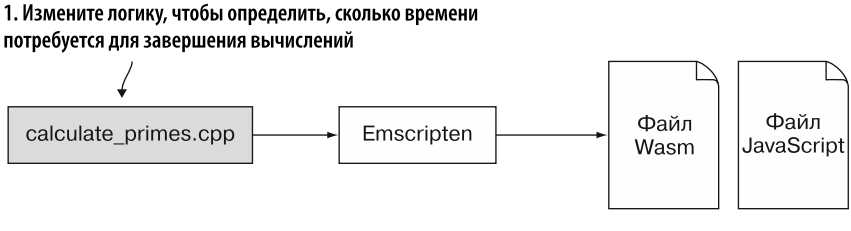


Рис. 9.5. Шаг 1 — изменение логики calculate_primes с целью определить, сколько времени займут вычисления

9.3.1. Изменение логики calculate_primes

Начнем. В папке WebAssembly\ создайте папку Chapter 9\9.3 pre-fetch\source\.

Скопируйте файл `calculate_primes.cpp` из папки Chapter 7\7.2.2 dlopen\source\ в только что созданную папку source\. Откройте этот файл в своем любимом редакторе.

Для этого сценария мы будем использовать класс `vector`, определенный в заголовочном файле `vector`, для хранения списка простых чисел, найденных в указанном диапазоне. Также мы будем применять класс `high_resolution_clock`, определенный в заголовочном файле `chrono`, чтобы установить время, которое необходимо коду для определения простых чисел.

Добавьте строки `include` для заголовочных файлов `vector` и `chrono` после включения `cstdio` в файле `calculate_primes.cpp`, как показано ниже:

```
#include <vector>
#include <chrono>
```

Теперь удалите объявление `EMSCRIPTEN_KEEPALIVE` над функцией `FindPrimes` — эта функция не будет вызываться извне модуля.

Вместо того чтобы вызывать `printf` для каждого найденного простого числа, изменим логику функции `FindPrimes` так, чтобы вместо этого добавить простое число к объекту `vector`. Это необходимо для определения продолжительности выполнения самих вычислений без задержки из-за вызова JavaScript-кода в каждом цикле. Затем изменим функцию `main`, чтобы она обрабатывала отправку информации о простых числах в окно консоли браузера.

ОПРЕДЕЛЕНИЕ

Объект vector — это контейнер последовательности для массивов динамического размера, в котором объем памяти автоматически увеличивается или уменьшается по мере необходимости. Более подробная информация об объекте vector доступна на странице https://en.cppreference.com/w/cpp/container/vector.

Внесите следующие изменения в функцию `FindPrimes`:

- добавьте параметр в функцию, чтобы она принимала ссылку на `std::vector<int>`;
- удалите все вызовы `printf`;
- в операторе `if` в `IsPrime` добавьте значение `i` к ссылке на вектор.

В файле `calculate_primes.cpp` измените функцию `FindPrimes` так, чтобы она соответствовала коду, показанному ниже:

```
void FindPrimes(int start, int end,
    std::vector<int>& primes_found) {
  for (int i = start; i <= end; i += 2) {
    if (IsPrime(i)) {
      primes_found.push_back(i);
    }
  }
}
```

Был добавлен параметр — ссылка на вектор

Простое число добавляется в список

Следующий шаг — изменение функции `main`:

- обновите окно консоли браузера, указав диапазон, в котором будет выполняться поиск простых чисел;
- определите, сколько времени требуется для выполнения функции `FindPrimes`, получив значение времени до и после вызова функции `FindPrimes` и рассчитав разницу между ними;
- создайте объект `vector` для хранения найденных простых чисел и передайте его в функцию `FindPrimes`;
- обновите консоль браузера, чтобы показать, сколько времени потребуется для выполнения функции `FindPrimes`;
- выведите все простые числа, которые были найдены путем перебора значений объекта `vector`.

Функция `main` в файле `calculate_primes.cpp` теперь должна соответствовать коду, показанному в листинге 9.1.

Листинг 9.1. Функция main в файле calculate_primes.cpp

```
...

int main() {
  int start = 3, end = 1000000;
  printf("Prime numbers between %d and %d:\n", start, end);

  std::chrono::high_resolution_clock::time_point duration_start =
      std::chrono::high_resolution_clock::now();

  std::vector<int> primes_found;
  FindPrimes(start, end, primes_found);

  std::chrono::high_resolution_clock::time_point duration_end =
      std::chrono::high_resolution_clock::now();

  std::chrono::duration<double, std::milli> duration =
      (duration_end - duration_start);
```

Получает текущее значение времени, отмечая начало выполнения FindPrimes

Создает объект vector для хранения целочисленных значений и передает его в функцию FindPrimes

Получает текущее значение времени, отмечая конец выполнения FindPrimes

Определяет время в миллисекундах, нужное для выполнения FindPrimes

```
  printf("FindPrimes took %f milliseconds to execute\n", duration.count());

  printf("The values found:\n");
  for(int n : primes_found) {
    printf("%d ", n);
  }
  printf("\n");

  return 0;
}
```

Перебирает все значения объекта vector и выводит их в консоль

После того как изменения в файл `calculate_primes.cpp` будут внесены, перейдем ко второму шагу (рис. 9.6) — генерации файлов WebAssembly с помощью Emscripten.

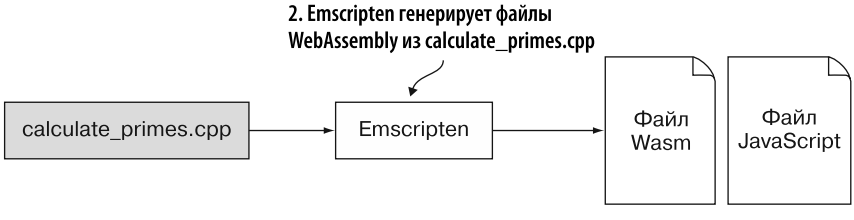


Рис. 9.6. Используйте Emscripten для генерации файлов WebAssembly из calculate_primes.cpp

9.3.2. Использование Emscripten для генерации файлов WebAssembly

Поскольку код на C++ в файле `calculate_primes.cpp` теперь использует chrono — одну из особенностей стандарта ISO C++ 2011, нужно сообщить Clang, компилятору клиентской части Emscripten, о необходимости применять этот стандарт, добавив параметр `-std=c++11`.

СПРАВКА

Emscripten использует Clang в качестве компилятора клиентской части, который принимает код на C++ и компилирует его в LLVM IR. По умолчанию Clang задействует стандарт C++98, но можно включить другие стандарты с помощью параметра -std. Clang поддерживает стандарты C++98/C++03, C++11, C++14 и C++17. Если это вас интересует, то на веб-странице https://clang.llvm.org/cxx_status.html вы можете найти более подробную информацию о стандартах C++, поддерживаемых Clang.

Кроме того, поскольку вы инициализируете объект `Module` в Emscripten после загрузки веб-страницы, также нужно добавить параметр `-s MODULARIZE=1`. Он сообщает Emscripten о необходимости заключить сгенерированный объект `Module`

файла JavaScript в функцию. Обертка в функцию предотвращает инициализацию объекта `Module` до тех пор, пока вы не создадите его экземпляр, что позволяет контролировать момент инициализации.

Чтобы скомпилировать файл `calculate_primes.cpp` в модуль WebAssembly, откройте командную строку, перейдите к папке Chapter 9\9.3 pre-fetch\source\ и выполните следующую команду:

```
emcc calculate_primes.cpp -O1 -std=c++11 -s MODULARIZE=1
➥ -o calculate_primes.js
```

9.3.3. Копирование в правильное место

После того как файлы WebAssembly будут созданы, можно сделать следующие шаги — скопировать эти файлы туда, где веб-сайт сможет их использовать (рис. 9.7). Затем создать файлы HTML и JavaScript для веб-страницы, которая будет создавать веб-воркер. Когда веб-страница получит скомпилированный модуль WebAssembly от веб-воркера, JavaScript в Emscripten будет использовать скомпилированный модуль, вместо того чтобы загружать его самостоятельно.

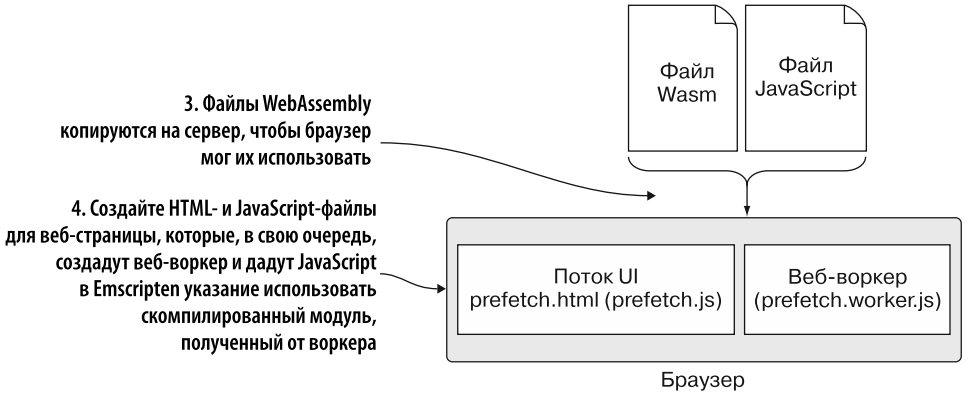


Рис. 9.7. Скопируйте файлы WebAssembly на сервер, чтобы браузер мог их использовать. Затем создайте HTML- и JavaScript-файлы для веб-страницы. JavaScript создаст веб-воркер, а затем даст JavaScript в Emscripten указание использовать скомпилированный модуль, полученный от веб-воркера

В папке Chapter 9\9.3 pre-fetch\ создайте папку frontend\, а затем скопируйте в нее следующие файлы:

- `calculate_primes.wasm` и `calculate_primes.js` из папки source\;
- `main.html` из папки Chapter 7\7.2.4 ManualLinking\frontend\; переименуйте файл в `prefetch.html`.

9.3.4. Создание HTML-файла для веб-страницы

Откройте файл `prefetch.html` из папки Chapter 9\9.3 pre-fetch\frontend\ в редакторе. Добавьте новый тег `script` перед текущим тегом `script` и присвойте его атрибуту `src` значение `prefetch.js` — файл JavaScript веб-страницы, который мы сейчас будем создавать.

Необходимо также изменить значение `src` другого тега `script` на `calculate_primes.js`, чтобы загрузить его в файл JavaScript, созданный Emscripten. Код файла `prefetch.html` теперь должен соответствовать коду, показанному в листинге 9.2.

Листинг 9.2. HTML-код в файле prefetch.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    HTML page I created for my WebAssembly module.

    <script src="prefetch.js"></script>
    <script src="calculate_primes.js"></script>
  </body>
</html>
```

Добавляет новый тег script для prefetch.js

Изменяет значение src на calculate_primes.js

9.3.5. Создание файла JavaScript для веб-страницы

Добавив HTML-код, перейдем к созданию файла JavaScript для веб-страницы. В папке Chapter 9\9.3 pre-fetch\frontend\ создайте файл prefetch.js и откройте его в своем любимом редакторе.

JavaScript-код должен будет выполнять следующие задачи.

1. Создать веб-воркер и подключить обработчик события `onmessage`:
   а) когда веб-воркер вызывает обработчик события `onmessage`, поместить полученный скомпилированный модуль в глобальную переменную;
   б) затем создать экземпляр объекта `Module` в Emscripten и указать функцию обратного вызова для функции `instantiateWasm` в Emscripten.
2. Определить функцию обратного вызова для функции `instantiateWasm` в Emscripten. При вызове ваша функция создаст экземпляр скомпилированного модуля, который содержится в глобальной переменной, и передаст созданный экземпляр модуля WebAssembly в код Emscripten.

СПРАВКА

Функция instantiateWasm вызывается кодом JavaScript в Emscripten в целях создания экземпляра модуля WebAssembly. По умолчанию JavaScript в Emscripten автоматически загружает и создает экземпляр модуля WebAssembly, но эта функция позволяет вам самостоятельно управлять этим процессом.

Первое, что потребуется вашему JavaScript-коду, — это пара глобальных переменных:

- одна будет содержать скомпилированный модуль, полученный от веб-воркера;
- другая будет содержать экземпляр JavaScript-объекта `Module` в Emscripten.

Добавьте переменные, показанные в следующем фрагменте кода, в файл `prefetch.js`:

```
let compiledModule = null;
let emscriptenModule = null;
```

Теперь вам нужно создать веб-воркер и подключить обработчик события `onmessage`, чтобы можно было получать сообщения от этого веб-воркера.

Создание веб-воркера и подключение обработчика события onmessage

Вы можете создать веб-воркер, создав экземпляр объекта `Worker`. Конструктор объекта `Worker` принимает путь к JavaScript-файлу, в котором хранится код воркера. В нашем случае это будет файл `prefetch.worker.js`.

Имея экземпляр объекта `Worker`, вы можете передавать ему сообщения, вызывая метод `postMessage` экземпляра. Вы также можете получать сообщения, подключив обработчик к событию `onmessage` экземпляра.

Создавая веб-воркер, вы настраиваете обработчик события `onmessage` для прослушивания сообщения от воркера. Когда событие вызывается, код помещает полученный скомпилированный модуль WebAssembly в глобальную переменную `compiledModule`.

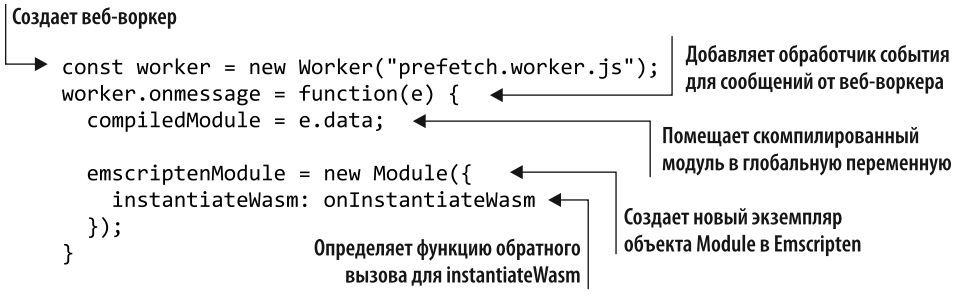
СПРАВКА

Обработчик события onmessage получит объект MessageEvent, который содержит данные, отправленные вызывающей стороной, в свойстве data. Объект MessageEvent является производным от объекта Event и представляет сообщение, полученное целевым объектом. Более подробную информацию об объекте MessageEvent можно найти на странице MDN Web Docs по адресу http://mng.bz/pyPw.

После этого ваш обработчик события `onmessage` создаст экземпляр JavaScript-объекта `Module` в Emscripten и определит функцию обратного вызова для функции

`instantiateWasm` Emscripten. Вы укажете эту функцию обратного вызова, чтобы переопределить обычное поведение Emscripten, и создадите экземпляр модуля WebAssembly из скомпилированного модуля, хранящегося в глобальной переменной.

Добавьте код, показанный в следующем фрагменте, в файл `prefetch.js`:



Теперь нужно реализовать функцию обратного вызова `onInstantiateWasm`, указанную для функции `instantiateWasm` в Emscripten.

Определение функции обратного вызова для функции instantiateWasm в Emscripten

Функция обратного вызова `instantiateWasm` принимает два параметра:

- `imports`:
  - этот параметр представляет собой `importObject`, который нужно передать в функцию создания экземпляра из API JavaScript в WebAssembly;
- `successCallback`:
  - после того как экземпляр модуля WebAssembly создан, необходимо передать его обратно в Emscripten с помощью этой функции.

Возвращаемое значение функции `instantiateWasm` зависит от того, создаете вы экземпляр модуля WebAssembly асинхронно или синхронно:

- если вы решите использовать асинхронную функцию, как в нашем случае, то возвращаемое значение должно быть пустым объектом JavaScript (`{}`);
- не рекомендуется задействовать синхронные вызовы API JavaScript в WebAssembly, если код выполняется в основном потоке браузера и даже может быть заблокирован некоторыми браузерами. Если используется синхронная функция, то возвращаемое значение должно быть объектом `exports` экземпляра модуля.

В данном случае вы не сможете использовать функцию `WebAssembly.instantiateStreaming` для создания экземпляра модуля WebAssembly, поскольку функция `instantiateStreaming` не принимает скомпилированный модуль. Вместо этого нужно будет использовать перегруженную функцию `WebAssembly.instantiate`:

- основная перегруженная функция `WebAssembly.instantiate` принимает байт-код двоичного файла WebAssembly в форме `ArrayBuffer`, а затем компилирует и создает экземпляр модуля. Когда промис разрешается, вам предоставляется объект, который содержит как `WebAssembly.Module` (скомпилированный модуль), так и объект `WebAssembly.Instance` (экземпляр);
- другая перегруженная функция `WebAssembly.instantiate` — та, которую мы будем здесь использовать. Она принимает объект `WebAssembly.Module` и создает его экземпляр. Когда промис разрешается, вам предоставляется только объект `WebAssembly.Instance`.

Добавьте код, показанный ниже, после обработчика события `onmessage` в файле `prefetch.js`:

```
function onInstantiateWasm(importObject, successCallback) {
  WebAssembly.instantiate(compiledModule,
      importObject).then(instance =>
    successCallback(instance)
  );

  return {};
}
```

Функция обратного вызова для instantiateWasm в Emscripten

Создает экземпляр скомпилированного модуля

Передает экземпляр модуля в функцию обратного вызова в Emscripten

Поскольку мы используем асинхронный вариант, возвращается пустой JavaScript-объект

Последний шаг, следующий за добавлением основного кода JavaScript веб-страницы, — создание JavaScript-кода веб-воркера (рис. 9.8). JavaScript получит и скомпилирует модуль WebAssembly, а затем передаст его в поток пользовательского интерфейса.

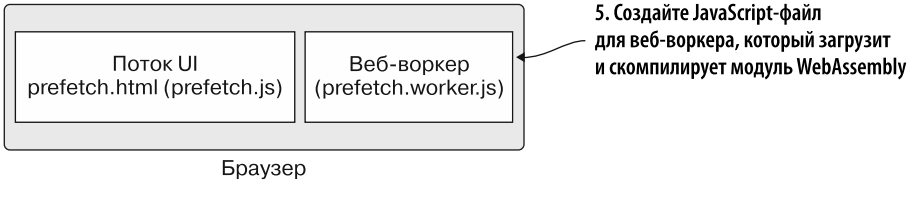


Рис. 9.8. Последний шаг — создание JavaScript-файла для веб-воркера, который загрузит и скомпилирует модуль WebAssembly. Тот будет передан в поток пользовательского интерфейса

9.3.6. Создание файла JavaScript для веб-воркера

В папке Chapter 9\9.3 pre-fetch\frontend\ создайте файл `prefetch.worker.js` и откройте его в своем любимом редакторе.

СОВЕТ

Название файла JavaScript не имеет значения, но это соглашение об именах ([название файла JavaScript, который создаст веб-воркер].worker.js) упрощает различие между обычными файлами JavaScript и теми, которые используются в веб-воркерах при беглом просмотре файловой системы. Оно также упрощает определение взаимосвязи между файлами, что поможет при отладке или поддержке кода.

Первое, что сделает код вашего веб-воркера, — загрузит и скомпилирует модуль `calculate_primes.wasm` в WebAssembly. Чтобы скомпилировать модуль, воспользуемся функцией `WebAssembly.compileStreaming`. После компиляции ваш код передаст модуль в поток UI, вызвав `postMessage` для своего глобального объекта `self`.

СПРАВКА

В потоке пользовательского интерфейса браузера глобальным объектом является объект окна. В веб-воркере глобальный объект — это self.

Добавьте код, показанный в следующем фрагменте, в файл `prefetch.worker.js`:

```
WebAssembly.compileStreaming(fetch("calculate_primes.wasm"))
.then(module => {
  self.postMessage(module);
});
```

Загружает и компилирует модуль WebAssembly

Передает скомпилированный модуль в основной поток

После того как будет создано все, что нужно, можно просмотреть результаты.

9.3.7. Просмотр результатов

Вы можете открыть браузер и ввести `http://localhost:8080/prefetch.html` в адресную строку, чтобы увидеть созданную веб-страницу. Если вы нажмете клавишу `F12`, чтобы открыть инструменты разработчика браузера (рис. 9.9), то в окне консоли должен отобразиться список найденных простых чисел. Вы также должны увидеть, сколько времени потребовалось для выполнения вычислений.

Предположим, нужно сократить время выполнения, необходимое для нахождения простых чисел от 3 до 1 000 000. Вы полагаете, что в качестве решения подойдет создание нескольких потоков pthread, каждый из которых будет параллельно обрабатывать меньший блок чисел.

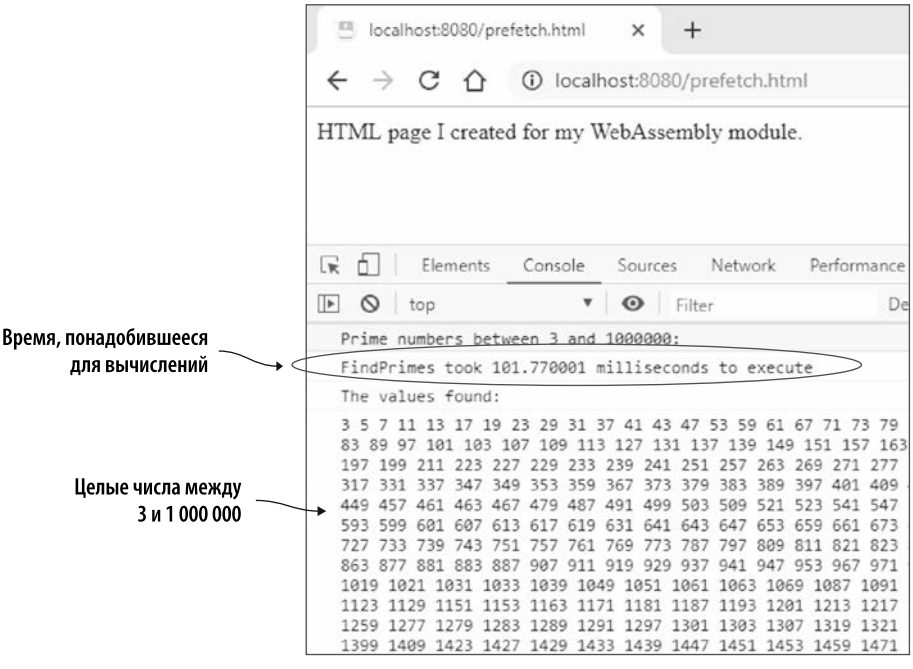


Рис. 9.9. Простые числа, найденные модулем WebAssembly, а также время, которое понадобилось для вычислений

9.4. ИСПОЛЬЗОВАНИЕ PTHREAD

WebAssembly поддерживает потоки pthread с помощью веб-воркеров и *SharedArrayBuffer*.

НАПОМИНАНИЕ

Поток — это путь выполнения внутри процесса; процесс может иметь несколько потоков. Поток pthread, также известный как поток POSIX, представляет собой API, определенный стандартом POSIX.1c для модуля выполнения, не зависящий от языка программирования (см. https://en.wikipedia.org/wiki/POSIX_Threads).

SharedArrayBuffer похож на ArrayBuffer, который обычно используется для памяти модуля WebAssembly. Разница в том, что SharedArrayBuffer позволяет разделять память модуля между основным модулем и каждым из его веб-воркеров. Он также позволяет выполнять *атомарные* операции для синхронизации памяти.

Поскольку память разделяется между модулем и его веб-воркерами, каждая область может читать и записывать одни и те же данные в памяти. Операции атомарного доступа к памяти обеспечивают следующее:

- записываются и читаются предсказуемые значения;
- текущая операция завершается до начала следующей;
- операции не прерываются.

Получить дополнительную информацию о поддержке потоков в WebAssembly, включая подробную информацию о различных вариантах доступа к атомарной памяти, можно на странице GitHub: http://mng.bz/O9xa.

ПРЕДУПРЕЖДЕНИЕ

Поддержка потоков для pthread в WebAssembly была приостановлена в январе 2018 года, поскольку производители браузеров отключили поддержку SharedArrayBuffer, чтобы предотвратить использование уязвимостей Spectre/Meltdown. Производители браузеров работают над решениями для закрытия уязвимостей в SharedArrayBuffer, однако на данный момент pthread доступны только в версии браузера Chrome для ПК или при добавлении флага в браузере Firefox. Вы узнаете, как это сделать, в подразделе 9.4.3.

Дополнительную информацию о поддержке pthread в Emscripten можно найти на странице https://emscripten.org/docs/porting/pthreads.html.

Порядок решения описанной ранее проблемы следующий (рис. 9.10).

1. Измените логику файла `calculate_primes` из раздела 9.3, чтобы создать четыре pthread. Каждому потоку будет предоставлен блок чисел для обработки и поиска простых чисел.
2. Используйте Emscripten для создания файлов WebAssembly с включенной поддержкой pthread. В этом случае будет использоваться HTML-шаблон Emscripten для просмотра результатов.

Первый шаг — изменение логики `calculate_primes`, для того чтобы создать четыре потока pthread, и выдача каждому потоку указания искать простые числа в определенном блоке чисел.

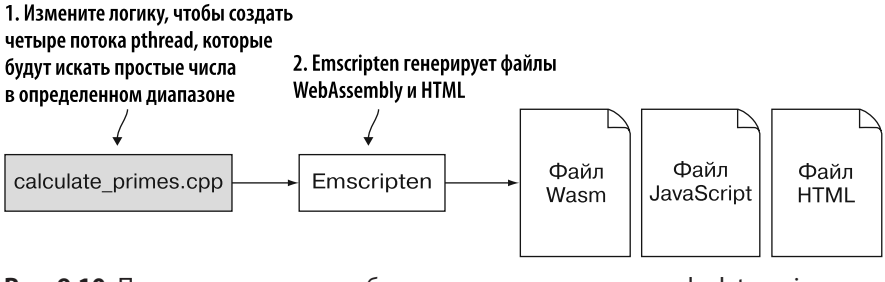


Рис. 9.10. Порядок решения проблемы — изменить логику calculate_primes, чтобы создать четыре потока pthread, и дать каждому из них указание искать простые числа в определенном диапазоне. Emscripten будет использоваться для генерации файлов WebAssembly и шаблона HTML

9.4.1. Изменение логики calculate_primes для создания и использования четырех pthread

В папке Chapter9\ создайте папку 9.4 pthreads\source\. Скопируйте файл `calculate_primes.cpp` из папки 9.3 pre-fetch\source\ в только что созданную папку source\ и откройте файл в своем любимом редакторе.

Поскольку вы будете использовать pthread, вам нужно добавить заголовочный файл `pthread.h` в файл `calculate_primes.cpp`, как показано в этом фрагменте кода:

```
#include <pthread.h>
```

Первая функция, которую нужно изменить, — `FindPrimes`.

Изменения в функции FindPrimes

В функцию `FindPrimes` нужно добавить строку кода, чтобы проверить, является ли указанное начальное значение нечетным числом. Если число четное, то его значение увеличивается на 1, чтобы цикл начинался с нечетного числа.

Функция `FindPrimes` в файле `calculate_primes.cpp` должна выглядеть следующим образом:

```
void FindPrimes(int start, int end,
    std::vector<int>& primes_found) {
  if (start % 2 == 0) { start++; }

  for (int i = start; i <= end; i += 2) {
    if (IsPrime(i)) {
      primes_found.push_back(i);
    }
  }
}
```

Четное значение увеличивается на 1, чтобы оно стало нечетным

Следующий шаг — создание функции для запуска потока pthread.

Создание функции для запуска потока pthread

Сейчас мы создадим функцию, которая будет использоваться в качестве процедуры запуска для каждого потока pthread. Эта функция, в свою очередь, будет вызывать функцию `FindPrimes`, но ей нужно будет знать начальные и конечные значения. Потребуется также получить объект `vector` для передачи в `FindPrimes` найденных простых чисел.

Функция для запуска pthread принимает только один параметр, поэтому мы определим передаваемый объект, который содержит все необходимые значения. Добавьте следующий код после функции `FindPrimes` в файле `calculate_primes.cpp`:

```
struct thread_args {
  int start;
  int end;
  std::vector<int> primes_found;
};
```

Теперь создадим функцию запуска для потока pthread. Функция запуска должна возвращать `void*` и принимать единственный параметр `void*` для переданных аргументов. При создании pthread вы передаете объект `thread_args` со значениями, которые необходимо передать функции `FindPrimes`.

Добавьте код, показанный в следующем фрагменте, после структуры `thread_args` в файле `calculate_primes.cpp`:

```
void* thread_func(void* arg) {
  struct thread_args* args = (struct thread_args*)arg;

  FindPrimes(args->start, args->end, args->primes_found);

  return arg;
}
```

Функция, которая будет вызываться при создании pthread

Приводит значение arg к указателю thread_args

Вызывает функцию FindPrimes, передавая значения, полученные из указателя args

Последние изменения, которые нам нужно внести, — в функции `main`.

Изменения в функции main

Теперь мы изменим функцию `main`, чтобы она создавала четыре потока pthread и говорила каждому, в каком диапазоне из 200 000 чисел нужно выполнить поиск. Чтобы создать pthread, будем вызывать функцию `pthread_create`, передавая следующие параметры:

- ссылку на переменную `pthread_t`, которая будет содержать идентификатор потока, если поток создается успешно;
- *атрибуты* создаваемого потока. В этом случае мы передаем `NULL`, чтобы использовать атрибуты по умолчанию;
- функцию запуска потока;
- значение, передаваемое в параметр функции запуска.

СПРАВКА

Объект атрибутов создается путем вызова функции pthread_attr_init, которая возвращает переменную pthread_attr_t, содержащую атрибуты по умолчанию. При наличии объекта атрибуты можно настраивать, вызывая различные функции pthread_attr. Закончив использовать объект атрибутов, вызовите функцию pthread_attr_destroy. На веб-странице https://linux.die.net/man/3/pthread_attr_init можно найти дополнительную информацию об объекте атрибутов pthread.

Создав потоки pthread, вы получите основной поток, который также будет вызывать функцию `FindPrimes` для проверки простых чисел от 3 до 199 999.

Когда вызов `FindPrimes` в основном потоке завершается, вы должны убедиться, что каждый pthread завершил свои вычисления, прежде чем переходить к выводу найденных значений в консоль. Чтобы основной поток ждал завершения каждого потока pthread, вызовите функцию `pthread_join`, передав в качестве первого параметра идентификатор потока, выполнения которого нужно дождаться. Второй параметр можно использовать для получения статуса завершения объединенного потока, но в этом случае он вам не нужен, поэтому достаточно передать `NULL`. Функции `pthread_create` и `pthread_join` вернут `0` (ноль) в случае успешного вызова.

Измените функцию `main` в файле `calculate_primes.cpp`, чтобы она соответствовала коду, показанному в листинге 9.3.

Листинг 9.3. Функция main в файле calculate_primes.cpp

```
...

int main() {
  int start = 3, end = 1000000;
  printf("Prime numbers between %d and %d:\n", start, end);

  std::chrono::high_resolution_clock::time_point duration_start =
      std::chrono::high_resolution_clock::now();

  pthread_t thread_ids[4];
  struct thread_args args[5];

  int args_index = 1;
  int args_start = 200000;

  for (int i = 0; i < 4; i++) {
    args[args_index].start = args_start;
    args[args_index].end = (args_start + 199999);

    if (pthread_create(&thread_ids[i],
      NULL,
      thread_func,
      &args[args_index])) {
      perror("Thread create failed");
      return 1;
    }

    args_index += 1;
    args_start += 200000;
  }

  FindPrimes(3, 199999, args[0].primes_found);
```

Идентификатор каждого созданного потока

Аргументы каждого потока, выполняющего вычисления, включая основной

Ноль пропускаем, чтобы основной поток мог использовать первый элемент args

Первый фоновый поток начнет вычисления с 200 000

Использует атрибуты потока по умолчанию

Задает начало и конец диапазона для вычислений текущего потока

Создает pthread. В случае успеха идентификатор потока будет помещен в массив по заданному индексу

Функция запуска потока

Аргументы для текущего потока

Увеличивает значения для следующего цикла

Использует основной поток для поиска простых чисел и помещения их в массив аргументов под индексом 0

```
    for (int j = 0; j < 4; j++) {
    pthread_join(thread_ids[j], NULL);
  }

  std::chrono::high_resolution_clock::time_point duration_end =
      std::chrono::high_resolution_clock::now();

  std::chrono::duration<double, std::milli> duration =
      (duration_end - duration_start);

  printf("FindPrimes took %f milliseconds to execute\n",
         duration.count());

  printf("The values found:\n");
  for (int k = 0; k < 5; k++) {
    for(int n : args[k].primes_found) {
      printf("%d ", n);
    }
  }
  printf("\n");

  return 0;
}
```

Указывает, что основному потоку нужно дождаться завершения всех pthread

Перебирает массив args

Перебирает список простых чисел из текущего элемента массива args

После того как изменения в файл `calculate_primes.cpp` будут внесены, можно сделать следующий шаг — дать Emscripten указание сгенерировать файлы WebAssembly и HTML (рис. 9.11).

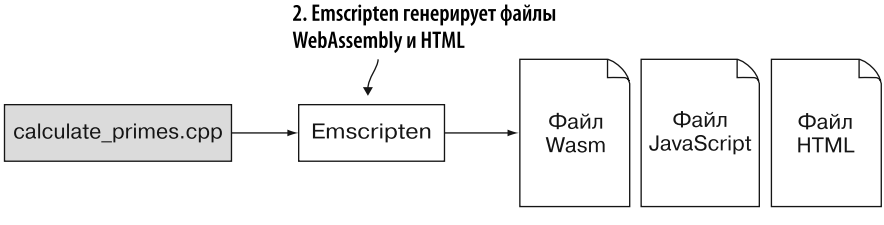


Рис. 9.11. Следующий шаг — использование Emscripten для генерации файлов WebAssembly и HTML из calculate_primes.cpp

9.4.2. Использование Emscripten для генерации файлов WebAssembly

Чтобы включить потоки pthread в модуле WebAssembly, нужно указать параметр `-s USE_PTHREADS=1` в командной строке при компиляции модуля. Необходимо также задать число потоков, которое планируется использовать одновременно, с помощью следующего параметра: `-s PTHREAD_POOL_SIZE=4`.

ПРЕДУПРЕЖДЕНИЕ

Если вы указываете значение больше 0 (ноль) для PTHREAD_POOL_SIZE, то все веб-воркеры для пула потоков будут добавлены при создании модуля, а не при вызове вашим кодом pthread_create. Запрашивая больше потоков, чем действительно нужно, вы потратите время обработки при запуске, а также часть памяти браузера для потоков, которые не будут ничего делать. Кроме того, рекомендуется протестировать модуль WebAssembly во всех браузерах, которые вы собираетесь поддерживать. Firefox указал, что поддерживает до 512 одновременных экземпляров веб-воркеров, но это количество может варьироваться в зависимости от браузера.

Если вы не укажете флаг `PTHREAD_POOL_SIZE`, то это будет означать то же самое, что указать флаг со значением `0` (ноль). С помощью данного подхода можно создать веб-воркеры при вызове `pthread_create`, а не во время создания экземпляра модуля. Однако при использовании этого метода выполнение потока не начинается немедленно. Вместо этого поток должен сначала вернуть выполнение браузеру. Ниже описан один из подходов к использованию этой функции:

- определите две функции в модуле: одну для вызова `pthread_create`, а другую — для `pthread_join`;
- JavaScript-коду сначала необходимо вызвать функцию, чтобы запустить `pthread_create`;
- затем JavaScript вызовет функцию `pthread_join` для получения результатов.

Чтобы скомпилировать модуль, откройте командную строку, перейдите в папку Chapter 9\9.4 pthreads\source\ и выполните следующую команду:

```
emcc calculate_primes.cpp -O1 -std=c++11 -s USE_PTHREADS=1
➥ -s PTHREAD_POOL_SIZE=4 -o pthreads.html
```

Вы могли заметить (это также изображено на рис. 9.12), что в папке появился файл с расширением `.mem`. Его необходимо распространить вместе с остальными сгенерированными файлами.

СПРАВКА

Файл .mem содержит сегменты данных для известного раздела «Данные» модуля, которые будут загружены в линейную память модуля при создании экземпляра. Наличие сегментов данных в отдельном файле позволяет создавать экземпляры модуля WebAssembly несколько раз, но загружать эти данные в память только однажды. Потоки pthread настроены таким образом, что каждый поток имеет собственный экземпляр модуля, но все модули используют одну и ту же память.

После того как файлы WebAssembly будут сгенерированы, можно просмотреть результаты.

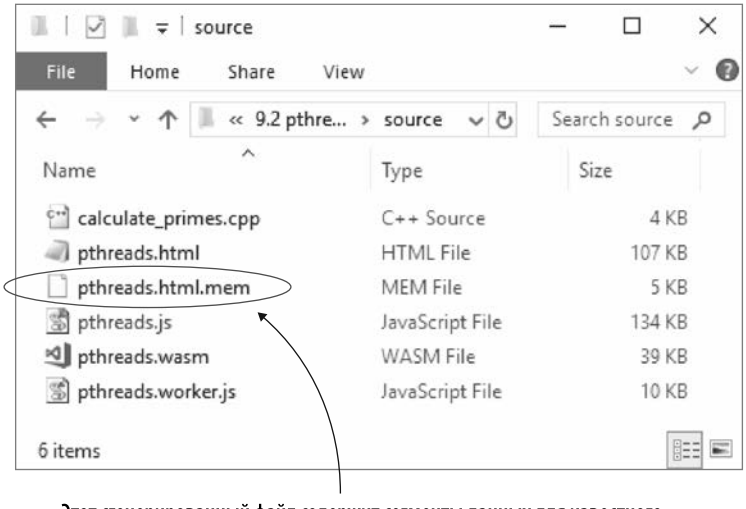


Рис. 9.12. Исходный файл calculate_primes.cpp и сгенерированные Emscripten файлы. В этом случае Emscripten помещает сегменты данных для известного раздела «Данные» модуля в отдельный файл

9.4.3. Просмотр результатов

На момент написания данной книги поддержка потоков в WebAssembly была доступна только для версии Chrome для ПК или при включении флага в Firefox. Флаг нужно включить перед просмотром сгенерированного файла `pthreads.html` в браузере Firefox.

Откройте браузер Firefox и напишите `about:config` в адресной строке. Должна открыться страница, похожая на изображенную на рис. 9.13. Нажмите кнопку I accept the risk! (Я беру на себя ответственность!), чтобы перейти к окну настроек.

Теперь вы должны увидеть страницу с длинным списком элементов. Чуть выше него находится поисковая строка. Введите `javascript.options.shared_memory` в поисковую строку — теперь список должен выглядеть как на рис. 9.14. Вы можете либо *дважды щелкнуть* на элементе списка, либо *щелкнуть правой кнопкой мыши* на элементе списка и выбрать Toggle (Переключить) в контекстном меню, чтобы изменить значение флага на `true`.

ПРЕДУПРЕЖДЕНИЕ

Этот параметр может быть отключен в Firefox из соображений безопасности. Завершив тестирование, вы должны вернуть этот флаг в значение false.

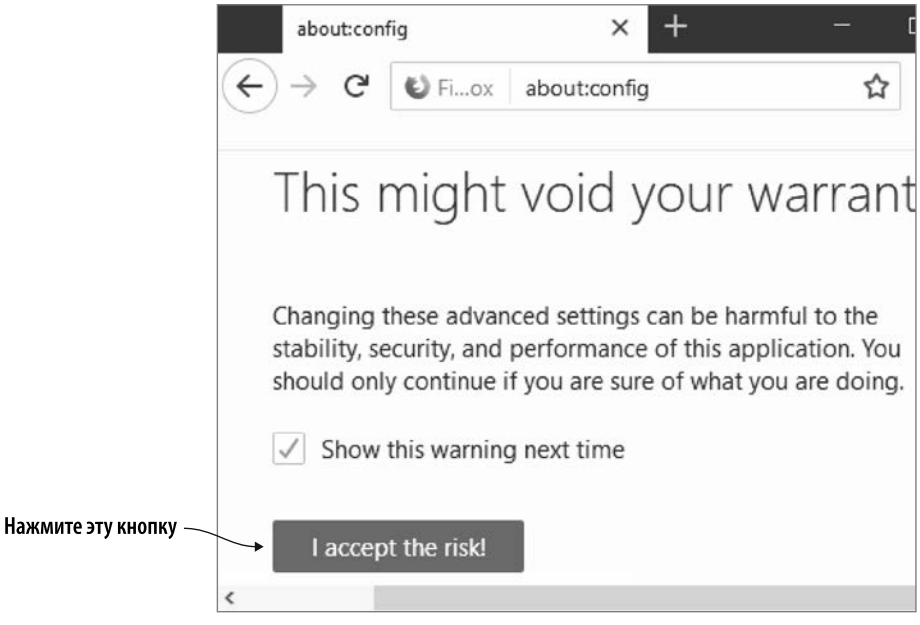


Рис. 9.13. Предупреждение перед настройками в Firefox. Нажмите кнопку I accept the risk! (Я беру на себя ответственность!), чтобы перейти к окну настроек

ПРИМЕЧАНИЕ

Было получено несколько сообщений о том, что Python SimpleHTTPServer не указывает правильный тип носителя для файлов JavaScript, используемых веб-воркерами. Он должен задействовать application/javascript, но в некоторых случаях вместо этого применяется text/plain. Если вы столкнулись с ошибками в Chrome, то попробуйте перейти на эту же веб-страницу в Firefox.

Чтобы просмотреть результаты, откройте окно браузера и введите `http://localhost:8080/pthreads.html` в адресную строку, чтобы увидеть созданную веб-страницу. Как показано на рис. 9.15, если вы нажмете клавишу `F12` для отображения инструментов разработчика браузера, то в окне консоли будет показан список найденных простых чисел, а также время, которое потребовалось для выполнения вычислений.

В подразделе 9.3.7 продолжительность поиска однопотоковым модулем WebAssembly простых чисел от 3 до 1 000 000 составила около 101 миллисекунды. Здесь использование четырех потоков pthread и основного потока для выполнения вычислений увеличило скорость выполнения почти в три раза.

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

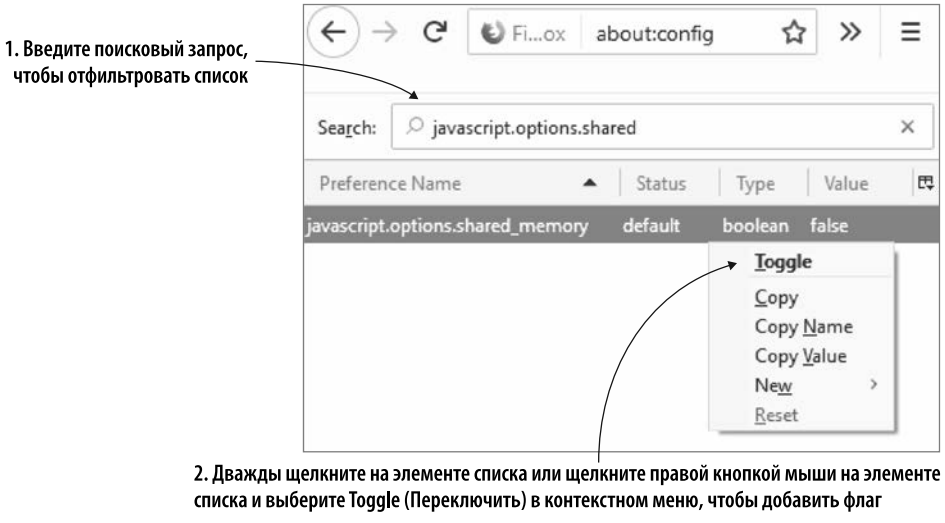


Рис. 9.14. Напишите javascript.options.shared_memory в поисковой строке, чтобы отфильтровать список. Дважды щелкните на элементе списка или щелкните правой кнопкой мыши на элементе списка и выберите Toggle (Переключить) в контекстном меню, чтобы изменить значение флага на true

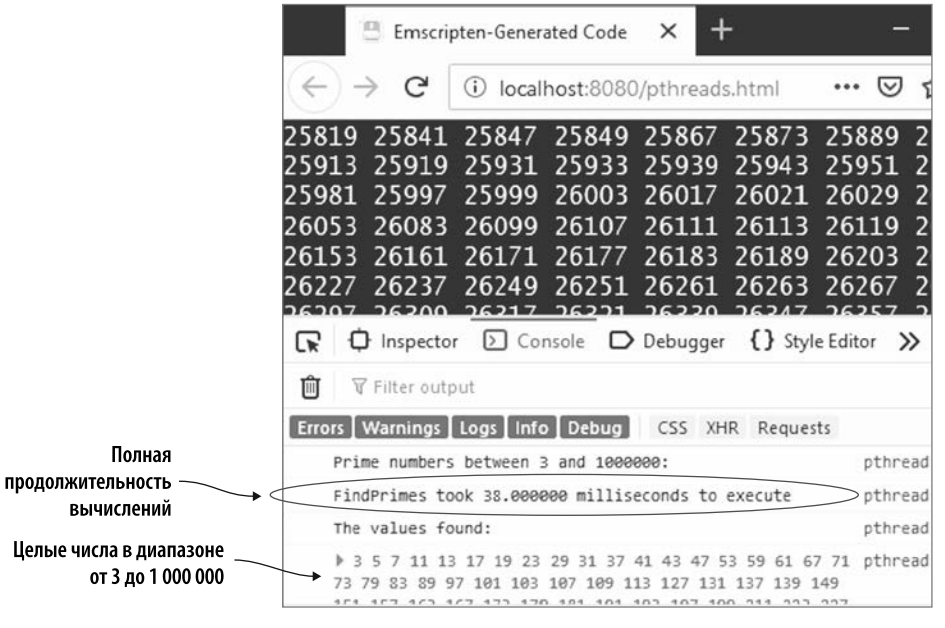


Рис. 9.15. Полное время выполнения для поиска простых чисел в диапазоне от 3 до 1 000 000 заняло 38 миллисекунд

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Потенциально использование веб-воркеров и потоков pthread открывает путь к ряду возможностей, от предварительной загрузки модулей WebAssembly до параллельной обработки. Ниже представлены некоторые из вариантов.

- Хотя веб-воркеры не заменяют pthread в WebAssembly полностью, они могут использоваться в качестве частичной замены для параллельной обработки в браузерах, которые еще не поддерживают pthread.
- Веб-воркеры могут служить для предварительной выборки и компиляции модулей WebAssembly по запросу. Это сокращает время загрузки, поскольку меньшее количество сущностей загружается и создается при первой загрузке веб-страницы, что делает ее более отзывчивой, поскольку страница тут же готова к взаимодействию с пользователем.
- В статье Пранава Джа и Сентила Падманабхана *WebAssembly at eBay: A Real-World Use Case* подробно рассказывается, как eBay использовал WebAssembly в сочетании с веб-воркерами и библиотекой JavaScript для улучшения своего сканера штрихкода: http://mng.bz/Ye1a.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Если бы вы хотели задействовать функцию из стандарта C++17, то какой флаг применили бы при компиляции модуля WebAssembly, чтобы сообщить Clang о необходимости использования этого стандарта?
2. Протестируйте изменение логики `calculate_primes` из раздела 9.4, используя три потока, а не четыре, чтобы увидеть, как это повлияет на продолжительность вычислений. Протестируйте тот же код с помощью пяти потоков и поместите вычисление основного потока в pthread, чтобы увидеть, влияет ли перенос всех вычислений из основного потока на их продолжительность.

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Если в основном потоке браузера происходит слишком много вычислений без периодического получения результатов, то пользовательский интерфейс может перестать отвечать. Если основной поток UI браузера не отвечает в течение достаточно долгого времени, то браузер может предложить пользователю завершить выполнение сценария.

- В браузерах существуют средства создания фоновых потоков, называемых веб-воркерами, а общение с веб-воркерами осуществляется путем передачи сообщений. Веб-воркеры не имеют доступа к DOM или другим аспектам пользовательского интерфейса браузера.
- Веб-воркеры можно применять для предварительной выборки ресурсов, которые могут понадобиться веб-странице в будущем, включая модули WebAssembly.
- Можно обрабатывать выборку и создание экземпляра модуля WebAssembly от имени JavaScript в Emscripten путем реализации функции обратного вызова `instantiateWasm`.
- Уже есть поддержка потоков pthread (потоков POSIX) в WebAssembly в Firefox, но может понадобиться включать флаг для их использования. Версия Chrome для ПК поддерживает pthreads без флага. Необходимо также скомпилировать модуль WebAssembly с помощью параметров командной строки `-s USE_PTHREADS` и `-s PTHREAD_POOL_SIZE` в Emscripten.
- Потоки pthread в WebAssembly используют веб-воркеры для потоковой обработки, SharedArrayBuffer в качестве разделяемой памяти между потоками и инструкции доступа к атомарной памяти для синхронизации взаимодействий с памятью.
- Все веб-воркеры для потоков pthread добавляются при создании экземпляра модуля WebAssembly, если при компиляции модуля указано значение флага командной строки `PTHREAD_POOL_SIZE`, равное `1` или больше. Если указано значение `0`, то поток pthread создается по запросу, но выполнение не начнется немедленно, если только поток сначала не вернет выполнение обратно браузеру.
- Можно дать Clang, компилятору внешнего интерфейса Emscripten, указание использовать стандарт C++, отличный от стандарта C++98, по умолчанию, добавив параметр командной строки `-std`.

10 Модули WebAssembly в Node.js

В этой главе

- ✓ Загрузка модуля WebAssembly с помощью кода JavaScript, сгенерированного Emscripten.
- ✓ Использование JavaScript API в WebAssembly для загрузки модуля WebAssembly.
- ✓ Работа с модулями WebAssembly, вызывающими JavaScript напрямую.
- ✓ Работа с модулями WebAssembly, использующими указатели функций для вызова JavaScript-функций.

В этой главе вы узнаете, как применять модули WebAssembly в Node.js. Последний слегка отличается от браузера — например, у него нет графического интерфейса пользователя (GUI), — но при работе с модулями WebAssembly вы найдете много общего между JavaScript, необходимым в браузере, и Node.js. Однако даже при таком сходстве рекомендуется протестировать модуль WebAssembly в Node.js, чтобы убедиться в его корректной работе в тех версиях, которые вы хотите поддерживать.

ОПРЕДЕЛЕНИЕ

Node.js — это среда выполнения JavaScript, построенная на движке V8 — том же самом движке, который работает и в браузере Chrome. Node.js позволяет использовать JavaScript в качестве серверного кода. Он также имеет большое количество доступных пакетов с открытым исходным кодом, помогающих решить многие проблемы программирования. От себя могу порекомендовать книгу, посвященную обучению Node.js, — «Node.js в действии, второе издание» (Manning): www.manning.com/books/node-js-in-action-second-edition.

Данная глава должна продемонстрировать, что WebAssembly можно использовать вне браузера. Желание применять его именно так привело к созданию стандартного интерфейса WebAssembly, или WASI, гарантирующего единообразие того, как хосты реализуют свои интерфейсы. Идея состоит в том, что модуль WebAssembly будет работать на любом хосте, поддерживающем WASI, — это могут быть периферийные вычисления, бессерверные хосты, хосты IoT (Internet of Things, Интернета вещей) и многие другие. Чтобы узнать больше о WASI, начните со статьи Саймона Биссона *Mozilla Extends WebAssembly Beyond the Browser with WASI, The New Stack* (http://mng.bz/E19R), в которой хорошо объясняются принципы.

10.1. ВСПОМИНАЯ ПРОЙДЕННОЕ

Вкратце вернемся к тому, что вы уже знаете. В главах с 4-й по 6-ю вы узнали о преимуществах повторного использования кода, которые дает WebAssembly. Мы изучили сценарий с применением уже существующего приложения для продаж для ПК, написанного на C++, которое нужно было перевести в онлайн-формат. Возможность повторного использования кода в нескольких средах снижает вероятность случайного внесения ошибок в случае необходимости поддерживать две или более версии одного и того же кода. Вдобавок повторное использование кода ведет к согласованности, когда логика работает одинаково во всех системах. Кроме того, поскольку существует только один источник кода для логики, меньшему количеству разработчиков нужно поддерживать его, что позволяет им работать над другими аспектами системы, а это, в свою очередь, повышает производительность.

Вы узнали, как изменить код на C++ так, чтобы его можно было скомпилировать в модуль WebAssembly с помощью компилятора Emscripten (рис. 10.1). Это позволило использовать один и тот же код как для приложения для ПК, так и в браузере. Затем вы узнали, как взаимодействовать с модулем WebAssembly в браузере, однако мы ни разу не обсуждали код сервера.

В этой главе вы узнаете, как загрузить модуль WebAssembly в Node.js. Вы также изучите, как модуль может вызывать JavaScript напрямую или с помощью указателей на функции.

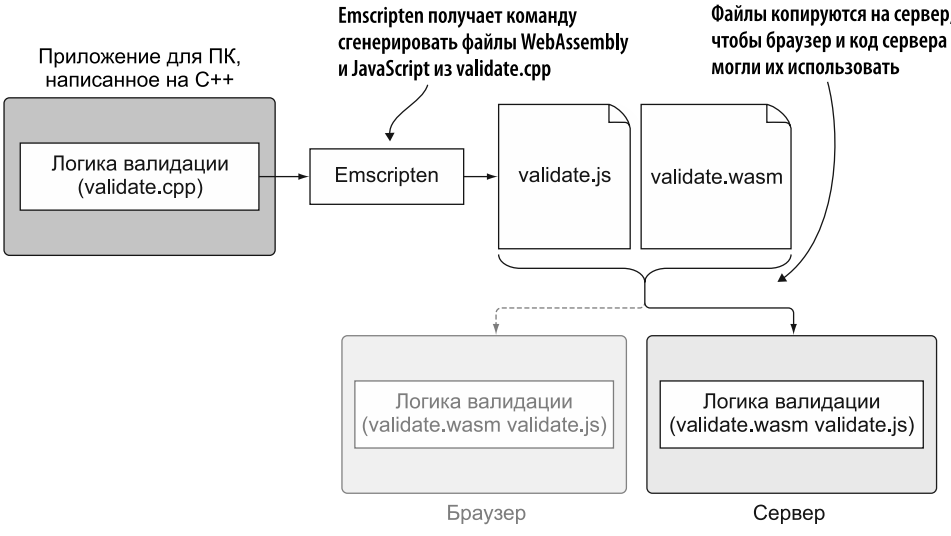


Рис. 10.1. Порядок превращения существующей логики на C++ в модуль WebAssembly в целях использования кодом сервера и браузером. Серверная часть обсуждается в этой главе

10.2. ВАЛИДАЦИЯ НА СТОРОНЕ СЕРВЕРА

Предположим, компания, которая создала онлайн-версию страницы Edit Product (Изменить товар) для своего приложения для продаж, теперь хочет передавать на сервер проверенные данные. Поскольку обойти проверку на стороне клиента (браузера) несложно, очень важно, чтобы код на стороне сервера проверял данные, полученные с сайта, прежде чем использовать их (рис. 10.2).

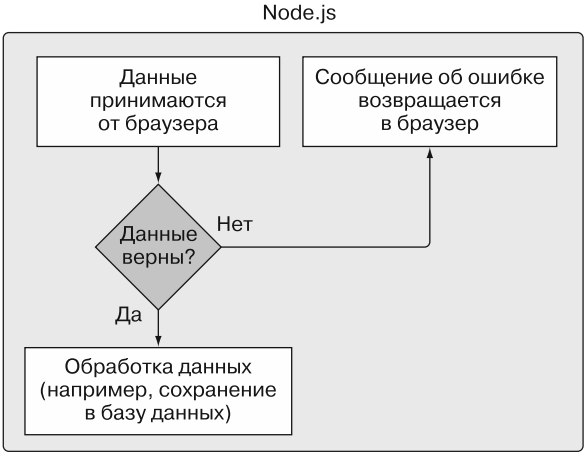


Рис. 10.2. Как работает валидация в Node.js

Логика веб-страницы на стороне сервера будет использовать Node.js, и поскольку последний поддерживает WebAssembly, вам не нужно заново создавать логику валидации. В этой главе мы будем применять те же самые модули WebAssembly, которые были созданы для использования в браузере в предыдущих главах. Это позволяет компании задействовать один и тот же код на C++ в трех местах: приложение для ПК, браузер и Node.js.

10.3. РАБОТА С МОДУЛЯМИ, СОЗДАННЫМИ EMSCRIPTEN

Как и при работе в браузере, в Node.js по-прежнему используется Emscripten в целях создания JavaScript-файлов для WebAssembly и Emscripten. Однако, в отличие от работы в браузере, HTML-файл создавать не нужно. Вместо этого, как показано в шаге 4 на рис. 10.3, вы создаете файл JavaScript, который загружает созданный Emscripten JavaScript-файл — тот затем будет самостоятельно обрабатывать загрузку и создание экземпляра модуля.

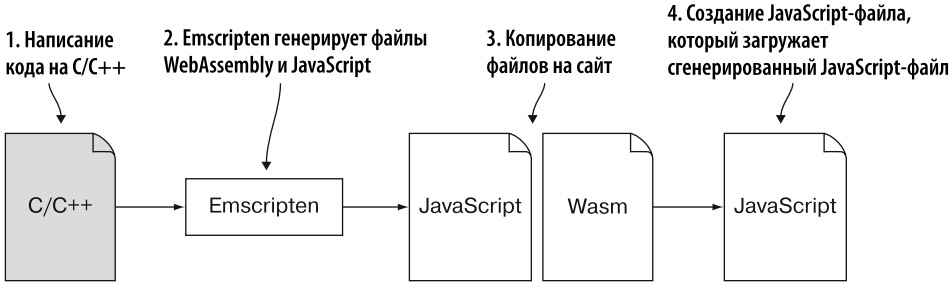


Рис. 10.3. Emscripten используется для генерации файлов JavaScript для WebAssembly. Затем вы создаете JavaScript-файл, который загружает JavaScript-файл, созданный Emscripten, а тот, в свою очередь, самостоятельно обрабатывает загрузку и создание экземпляра модуля

Способ подключения самого JavaScript, созданного Emscripten, в Node.js отличается по сравнению с браузером:

- *в браузере* код JavaScript из Emscripten подключается путем добавления ссылки на файл JavaScript в качестве тега `script` в файле HTML;
- *в Node.js* для загрузки файлов JavaScript используется функция `require` с путем к файлу, который нужно загрузить, в качестве параметра.

Задействовать файл JavaScript, созданный Emscripten, удобно, поскольку в коде JavaScript есть проверки, которые определяют, применяется ли он в браузере или

в Node.js; он загружает и создает экземпляр модуля в соответствии с окружающей средой, в которой используется. Все, что вам нужно сделать, — загрузить файл, а остальное сделает код.

Посмотрим, как добавить сгенерированный Emscripten файл JavaScript.

10.3.1. Загрузка модуля WebAssembly

В этом подразделе вы узнаете, как загрузить созданный Emscripten файл JavaScript, чтобы он затем самостоятельно мог загрузить и создать экземпляр вашего модуля WebAssembly. В папке WebAssembly\ создайте папку Chapter 10\10.3.1 JsPlumbingPrimes\backend\ для файлов, которые будут использоваться в этом подразделе. Скопируйте файлы `js_plumbing.wasm` и `js_plumbing.js` из папки Chapter 3\3.4 js_plumbing\ в только что созданную папку backend\.

В папке backend\ создайте файл `js_plumbing_nodejs.js` и откройте его в своем любимом редакторе. В данном файле мы добавим вызов функции `require` в Node.js, передав путь к файлу `js_plumbing.js`, созданному Emscripten. При загрузке с помощью Node.js код JavaScript Emscripten обнаружит, что он используется в Node.js, и автоматически загрузит и создаст экземпляр модуля WebAssembly `js_plumbing.wasm`.

Добавьте в файл `js_plumbing_nodejs.js` код, показанный ниже:

```
require('./js_plumbing.js');
```
← Подключает связующий файл Emscripten

Просмотр результатов

Чтобы дать Node.js указание запускать JavaScript, используйте окно консоли для запуска команды node, а затем укажите файл JavaScript, который нужно запустить. Чтобы открыть только что созданный файл `js_plumbing_nodejs.js`, запустите командную строку, перейдите в папку Chapter 10\10.3.1 JsPlumbingPrimes\backend\ и затем выполните следующую команду:

```
node js_plumbing_nodejs.js
```

Как показано на рис. 10.4, вы можете видеть, что модуль был загружен и запущен, поскольку в окне консоли отображаются выходные данные модуля: Prime numbers between 3 and 100000 (Простые числа от 3 до 100 000), за которыми следует список простых чисел, найденных в этом диапазоне.

Теперь вы знаете, как загрузить JavaScript-файл, сгенерированный Emscripten, в Node.js. Посмотрим, как вызывать функции в модуле WebAssembly в случае использования Node.js.

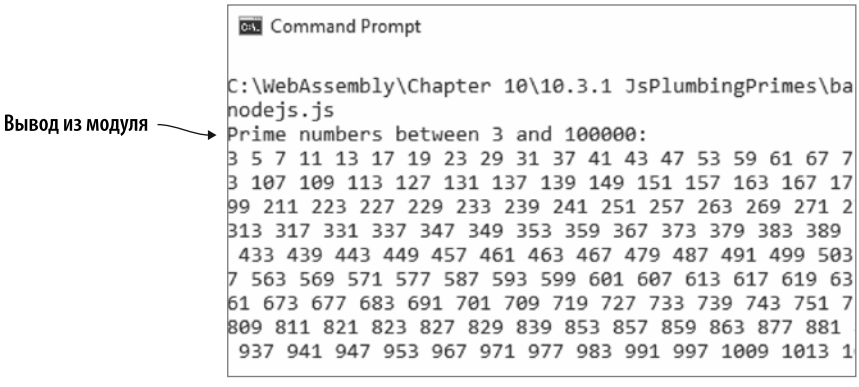


Рис. 10.4. Вывод в консоли из модуля WebAssembly в Node.js

10.3.2. Вызов функций в модуле WebAssembly

В главе 4 вы совершили некоторые шаги (рис. 10.5), чтобы вывести приложение для продаж на ПК в онлайн-формат. После того как веб-страница проверит

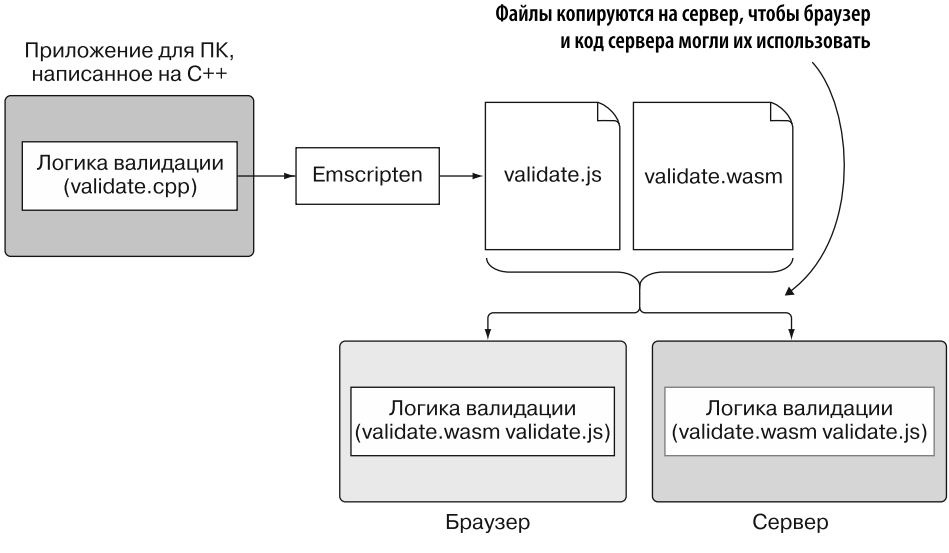


Рис. 10.5. Заключительный этап процесса повторного использования кода на C++ — серверная часть — в нашем случае Node.js. Скопируем сгенерированные файлы WebAssembly туда, где находятся файлы Node.js, а затем создадим файл JavaScript для взаимодействия с модулем

введенные пользователем данные, они отправляются на сервер, чтобы их можно было сохранить в базе данных или как-то обработать. Прежде чем код сервера сделает что-либо с полученными данными, он должен убедиться, что они правильны, поскольку существуют способы обойти проверку браузера. В данном случае сервер — это Node.js, и мы будем использовать тот же модуль WebAssembly, что и в браузере, в целях проверки полученных данных.

Теперь реализуем заключительный этап процесса вывода приложения для продаж на ПК в онлайн-формат, реализовав его серверную часть. Скопируем сгенерированные файлы WebAssembly туда, где находятся файлы Node.js, а затем создадим файл JavaScript для взаимодействия с модулем.

Создание серверного кода для Node.js

В папке WebAssembly\ создайте папку Chapter 10\10.3.2 JsPlumbing\backend\ для хранения файлов, которые будут использоваться в этом подразделе, а затем выполните следующие действия:

- скопируйте `validate.js`, `validate.wasm` и `editproduct.js` из папки Chapter 4\4.1 js_plumbing\frontend\ в только что созданную папку backend\;
- переименуйте файл `editproduct.js` в `nodejs_validate.js`, а затем откройте его в своем любимом редакторе.

Вместо того чтобы получать данные с веб-страницы, будем имитировать получение данных с помощью объекта `InitialData`, который переименуем в `clientData`. В файле `nodejs_validate.js` переименуйте объект `InitialData` в `clientData` следующим образом:

```
const clientData = {
  name: "Women's Mid Rise Skinny Jeans",
  categoryId: "100",
};
```

Объект, имитирующий данные, полученные из браузера

В целом JavaScript, требующийся Node.js, похож на тот, который мы использовали в браузере. Основное отличие кода Node.js состоит в том, что в нем нет UI, поэтому нет элементов управления вводом, с которыми нужно взаимодействовать. Следовательно, отдельные вспомогательные функции не понадобятся. Удалите следующие функции из файла `nodejs_validate.js`:

- `initializePage`;
- `getSelectedCategoryId`.

Поскольку пользовательского интерфейса нет, то нет и элемента для отображения сообщений об ошибках, полученных от модуля. Вместо этого будем выводить

сообщения об ошибках в консоль. Измените функцию `setErrorMessage`, чтобы вызвать `console.log`, как показано ниже:

```
function setErrorMessage(error) { console.log(error); }
```

В Node.js нет пользовательского интерфейса, поэтому все сообщения об ошибках будем выводить в консоль

Одно из различий между работой с файлом JavaScript, созданным Emscripten в Node.js, и работой в браузере состоит в том, что в браузере код JavaScript имеет доступ к глобальному объекту `Module`, но многие вспомогательные функции также находятся в глобальной области. В браузере такие функции, как `_malloc`, `_free` и `UTF8ToString`, находятся в глобальной области видимости и могут быть вызваны напрямую, без добавления к ним префикса `Module`, например `Module.malloc`. Однако в Node.js возвращаемый объект из вызова `require` — это объект `Module`, и все вспомогательные методы Emscripten доступны только через данный объект.

СОВЕТ

Вы можете назвать объект, который возвращается из функции require, как хотите. Поскольку здесь используется тот же код JavaScript, что и в браузере, проще задействовать имя Module, поэтому вам не придется изменять много JavaScript-кода. Если вы все же решите применить другое имя, то нужно будет изменить места в коде, которые вызывают Module.ccall, например, чтобы использовать имя вашего объекта вместо Module.

В файле `nodejs_validate.js` после функции `setErrorMessage` добавьте вызов функции `require` Node.js, чтобы загрузить файл JavaScript, созданный Emscripten (`validate.js`). Назовите объект, полученный от функции `require`, `Module`. Строка кода должна выглядеть так:

```
const Module = require('./validate.js');
```

Загружает сгенерированный с помощью Emscripten JavaScript-объект и называет его Module

Создание экземпляра модуля WebAssembly происходит асинхронно как в браузере, так и в Node.js. Чтобы получать уведомление, когда код JavaScript в Emscripten готов к взаимодействию, определите функцию `onRuntimeInitialized`.

В файле `nodejs_validate.js` преобразуйте функцию `onClickSave` в функцию свойства `onRuntimeInitialized` объекта `Module`. Кроме того, измените код функции, чтобы больше не извлекать `name` и `categoryId` из элементов управления, а вместо этого использовать объект `clientData`. Функция `onClickSave` в файле `nodejs_validate.js` теперь должна соответствовать коду, показанному в листинге 10.1.

Листинг 10.1. Функция onClickSave изменена и переименована в onRuntimeInitialized

```
...

Module['onRuntimeInitialized'] = function() {
  let errorMessage = "";
  const errorMessagePointer = Module._malloc(256);

  if (!validateName(clientData.name, errorMessagePointer) ||
      !validateCategory(clientData.categoryId,
          errorMessagePointer)) {
    errorMessage = Module.UTF8ToString(errorMessagePointer);
  }

  Module._free(errorMessagePointer);

  setErrorMessage(errorMessage);
  if (errorMessage === "") {

  }
}

...
```

Изменяет onClickSave на onRuntimeInitialized

Проверяет свойство name в объекте clientData

Проверяет свойство categoryId в объекте clientData

Проблем не обнаружено. Данные могут быть сохранены

Никаких других изменений в файле `nodejs_validate.js` не требуется.

Просмотр результатов

Если вы запустите код прямо сейчас, то проблем с валидацией не будет, поскольку все данные в объекте `clientData` правильные. Чтобы проверить логику валидации, вы можете изменить данные в объекте `clientData`, передав пустое значение для свойства `name` (`name: ""`), сохранив файл и запустив код.

Чтобы запустить файл JavaScript в Node.js, откройте командную строку, перейдите к папке Chapter 10\10.3.2 JsPlumbing\backend\, а затем выполните следующую команду:

```
node nodejs_validate.js
```

Должно появиться сообщение валидации, показанное на рис. 10.6.

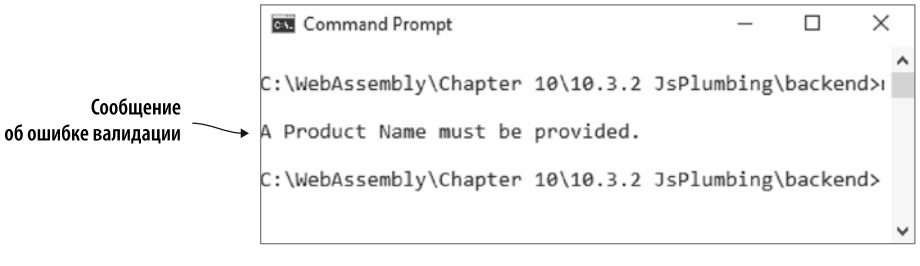


Рис. 10.6. Ошибка валидации в Node.js

Узнав, как загрузить файл JavaScript, сгенерированный Emscripten, в Node.js и вызвать функции в модуле WebAssembly, перейдем к вопросу, как запущенный в Node.js модуль может вызывать функции в JavaScript-файле.

10.3.3. Вызов JavaScript-функций

Как вы видели в предыдущем подразделе, функция может вызывать модуль и ждать ответа. Хотя этот подход работает, бывают случаи, когда модулю может потребоваться вызвать JavaScript напрямую, после того как он завершит выполнение некой работы, — возможно, чтобы получить дополнительную информацию или в целях обновления.

В модуль WebAssembly, который вы будете использовать в этом подразделе, включена функция из созданного Emscripten JavaScript-файла. Модуль вызовет данную функцию, если произошла ошибка, передав указатель на сообщение о ней. Функция прочитает данное сообщение из памяти модуля, а затем передаст строку в функцию `setErrorMessage` в основном JavaScript-файле.

Реализация серверного кода для Node.js

В папке WebAssembly\ создайте папку Chapter 10\10.3.3 EmJsLibrary\backend\ для хранения файлов, которые будут использоваться в этом подразделе, а затем выполните следующие действия:

- скопируйте файлы `validate.js`, `validate.wasm` и `editproduct.js` из папки Chapter 5\5.1.1 EmJsLibrary\frontend\ в только что созданную папку backend\;
- переименуйте файл `editproduct.js` в `nodejs_validate.js`, а затем откройте его в своем любимом редакторе.

В файле `nodejs_validate.js` переименуйте объект `InitialData` в `clientData`, как показано ниже:

```
const clientData = {    <-- Переименован из InitialData
  name: "Women's Mid Rise Skinny Jeans",
  categoryId: "100",
};
```

Удалите следующие функции из файла `nodejs_validate.js`:

- `initializePage`;
- `getSelectedCategoryId`.

Как оказалось, этот конкретный сценарий использования собственного JavaScript-кода в файле JavaScript, созданном Emscripten, неидеален при использовании

Node.js. Это связано с тем, что функция `require`, с помощью которой загружается файл JavaScript, помещает код из данного файла в собственную область видимости, вследствие чего код в созданном Emscripten файле JavaScript не может получить доступ ни к одной из функций в области родительского объекта (кода, который его загрузил). Ожидается, что код JavaScript, загруженный функцией `require`, будет самодостаточным и не станет вызывать функции из области видимости родительского объекта.

Если модулю необходимо вызвать что-то из области видимости родительского элемента, то лучше использовать указатель на функцию, который передает родительский элемент, как будет показано ниже. Но в этом случае, чтобы обойти проблему сгенерированного `validate.js` кода, который не может получить доступ к функции `setErrorMessage`, вам нужно будет создать функцию `setErrorMessage` как свойство для объекта `global`, а не как обычную функцию.

ИНФОБОКС

В браузерах областью видимости верхнего уровня является глобальная область (объект window). Однако в Node.js область верхнего уровня — это не глобальная область, а сам модуль. По умолчанию все переменные и объекты локальны для модуля в Node.js. Объект global представляет глобальную область видимости в Node.js.

Чтобы сделать функцию `setErrorMessage` доступной для сгенерированного Emscripten кода JavaScript, необходимо изменить ее так, чтобы она была частью объекта `global`, как показано ниже. Выведите сообщение об ошибке в консоль, заменив содержимое функции вызовом `console.log`:

```
global.setErrorMessage = function(error) {
  console.log(error);
}
```

Создает функцию в объекте global

Выводит сообщения об ошибке в консоль

После функции `setErrorMessage` добавьте вызов функции `require` из Node.js, чтобы загрузить файл JavaScript, созданный Emscripten (`validate.js`). Строка кода должна выглядеть так:

```
const Module = require('./validate.js');
```

Загружает файл JavaScript, сгенерированный Emscripten, и задает результирующему объекту имя Module

В файле `nodejs_validate.js` преобразуйте функцию `onClickSave` в функцию как свойство `onRuntimeInitialized` объекта `Module`. Затем измените код в функции, чтобы больше не вызывать функцию `setErrorMessage` и не извлекать `name` и `categoryId` из элементов управления. Наконец, используйте объект `clientData` для передачи `name` и `categoryId` функциям проверки.

Измененная функция `onRuntimeInitialized` должна соответствовать коду, показанному в следующем фрагменте:

```
Module['onRuntimeInitialized'] = function() {
  if (validateName(clientData.name) &&
      validateCategory(clientData.categoryId)){

  }
}
```

Изменяет onClickSave на onRuntimeInitialized

Проверяет свойство name в объекте clientData

Проверяет свойство categoryId в объекте clientData

Проблем не обнаружено. Данные могут быть сохранены

Никаких других изменений в файле `nodejs_validate.js` не требуется.

Просмотр результатов

Чтобы протестировать логику валидации, вы можете подправить данные в объекте `clientData`, изменив свойство `name` или `categoryId` на недопустимое значение. Например, измените `categoryId` на значение, которого нет в массиве `VALID_CATEGORY_IDS` (`categoryId: "1001"`), и сохраните файл.

Чтобы запустить файл JavaScript в Node.js, откройте командную строку, перейдите в папку Chapter 10\10.3.3 EmJsLibrary\backend\ и выполните следующую команду:

```
node nodejs_validate.js
```

Должно появиться сообщение проверки, как на рис. 10.7.

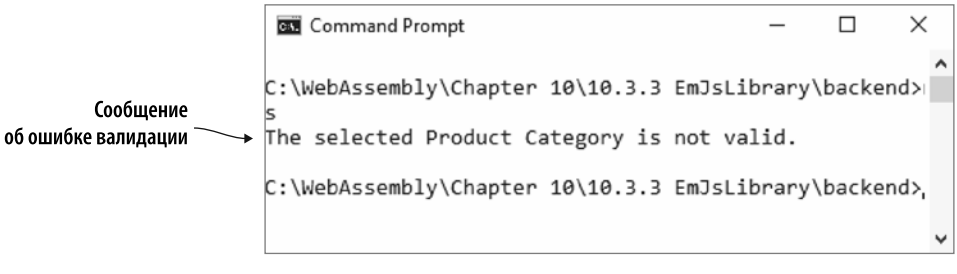


Рис. 10.7. Ошибка валидации категории товара в Node.js

Использование библиотеки JavaScript в Emscripten с кодом, который вызывает основной JavaScript-код приложения, не всегда может подойти, если вы планируете задействовать Node.js, из-за проблем с областью действия функции `require`. Если вы добавите пользовательский JavaScript в сгенерированный Emscripten файл JavaScript, который будет применяться в Node.js, то лучше всего сделать код самодостаточным, а не вызывать родительский код.

Если модулю WebAssembly необходимо вызвать основной JavaScript приложения при поддержке Node.js, то рекомендуется использовать указатели на функции, о которых вы узнаете ниже.

10.3.4. Вызов указателей на функции JavaScript

Возможность вызывать JavaScript напрямую полезна, но ваш JavaScript должен предоставлять функцию во время создания экземпляра модуля. После того как эта функция будет передана модулю, вы не можете заменить ее. В большинстве случаев это допустимо, но бывают варианты, когда удобно было бы передавать модулю функцию так, чтобы выполнять вызов по мере необходимости.

Реализация серверного кода для Node.js

В папке WebAssembly\ создайте папку Chapter 10\10.3.4 EmFunctionPointers\backend\ для хранения файлов, которые будут использоваться в этом подразделе, а затем выполните следующие действия:

- скопируйте файлы `validate.js`, `validate.wasm` и `editproduct.js` из папки Chapter 6\6.1.2 EmFunctionPointers\frontend\ в только что созданную папку backend\;
- переименуйте файл `editproduct.js` в `nodejs_validate.js`, а затем откройте его в своем любимом редакторе.

В файле `nodejs_validate.js` переименуйте объект `InitialData` в `clientData`, как показано ниже:

```
const clientData = {
  name: "Women's Mid Rise Skinny Jeans",
  categoryId: "100",
};
```

Объект, имитирующий данные, полученные из браузера

Удалите следующие функции из файла `nodejs_validate.js`:

- `initializePage`;
- `getSelectedCategoryId`.

Измените функцию `setErrorMessage`, чтобы вызывать `console.log`, как показано ниже:

```
function setErrorMessage(error) { console.log(error); }
```

В Node.js нет пользовательского интерфейса, поэтому мы будем выводить сообщения об ошибках в консоль

После функции `setErrorMessage` добавьте вызов функции `require` в Node.js, чтобы загрузить файл `validate.js`. Строка кода должна соответствовать следующему фрагменту:

```
const Module = require('./validate.js');
```

Загружает сгенерированный Emscripten файл JavaScript и задает возвращаемому объекту имя Module

В файле `nodejs_validate.js` преобразуйте функцию `onClickSave` в функцию как свойство `onRuntimeInitialized` объекта `Module`. Измените код в функции, чтобы больше не вызывать функцию `setErrorMessage` и не извлекать `name` и `categoryId` из элементов управления. Затем используйте объект `clientData` для передачи `name` и `categoryId` функциям проверки.

Измененная функция `onClickSave` должна теперь соответствовать коду, показанному в листинге 10.2.

Листинг 10.2. Функция onClickSave, измененная на onRuntimeInitialized

```
...

Module['onRuntimeInitialized'] = function() {
  Promise.all([
      validateName(clientData.name),
      validateCategory(clientData.categoryId)
  ])
  .then(() => {

  })
  .catch((error) => {
    setErrorMessage(error);
  });
}
```

Заменяет onClickSave на onRuntimeInitialized

Проверяет свойство name в объекте clientData

Проверяет свойство categoryId в объекте clientData

Проблем не обнаружено. Данные могут быть сохранены

Никаких других изменений в файле `nodejs_validate.js` не требуется.

Просмотр результатов

Чтобы протестировать логику валидации, вы можете подправить данные в объекте `clientData`, изменив свойство `name` на значение, превышающее значение `MAXIMUM_NAME_LENGTH`, равное 50 символам (`name: "This is a very long product name to test the validation logic."`), и сохранив файл.

Чтобы запустить файл JavaScript в Node.js, откройте командную строку, перейдите в папку Chapter 10\10.3.4 EmFunctionPointers\backend\ и выполните следующую команду:

```
node nodejs_validate.js
```

Должно появиться сообщение валидации, показанное на рис. 10.8.

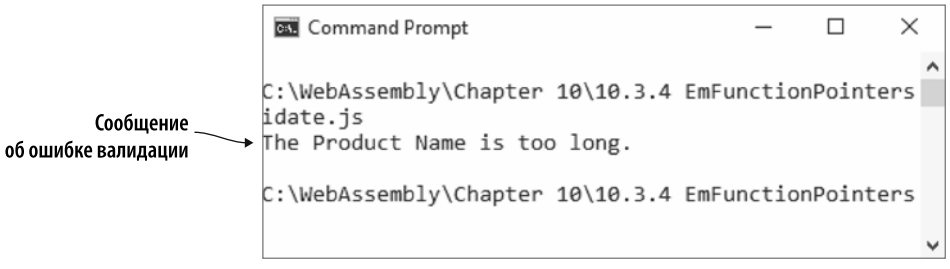


Рис. 10.8. Сообщение валидации названия товара в Node.js

Вы уже изучили, как работать с модулями WebAssembly в Node.js в случае, когда эти модули созданы с помощью сгенерированного Emscripten JavaScript-кода. Оставшаяся часть данной главы посвящена использованию модулей WebAssembly в Node.js, если эти модули созданы без сгенерированного Emscripten JavaScript-файла.

10.4. ИСПОЛЬЗОВАНИЕ JAVASCRIPT API В WEBASSEMBLY

В случае применения компилятора Emscripten производственный код обычно включает сгенерированный Emscripten файл JavaScript. Этот файл обрабатывает загрузку модуля WebAssembly и взаимодействие с JavaScript API в WebAssembly самостоятельно. Он также содержит ряд вспомогательных функций, облегчающих взаимодействие с модулем.

Отсутствие созданного файла JavaScript полезно в целях обучения, поскольку дает вам возможность загрузить файл `.wasm` и напрямую работать с JavaScript API в WebAssembly. Вы создаете объект JavaScript, содержащий значения и функции, которые модулю предстоит импортировать, а затем используете API для компиляции и создания экземпляра модуля. После того как он будет создан, вы получите доступ к экспорту модуля, что позволит вам взаимодействовать с модулем.

По мере расширения использования WebAssembly вполне вероятно, что будет создано множество сторонних модулей для расширения возможностей браузера. Кроме того, знание того, как работать с модулями, которые не используют код JavaScript от Emscripten, будет полезно, если вам когда-нибудь понадобится применить сторонний модуль, созданный с помощью компилятора, отличного от Emscripten.

В главах 3–6 мы использовали Emscripten в целях создания одного лишь файла `.wasm` с помощью параметра `SIDE_MODULE`. В результате был создан только модуль, без стандартных функций библиотеки C и без файла JavaScript в Emscripten. Поскольку файл JavaScript не был сгенерирован, теперь нужно добавить JavaScript, который необходим для загрузки и создания экземпляра модуля с помощью JavaScript API в WebAssembly, как показано в шаге 4 на рис. 10.9.

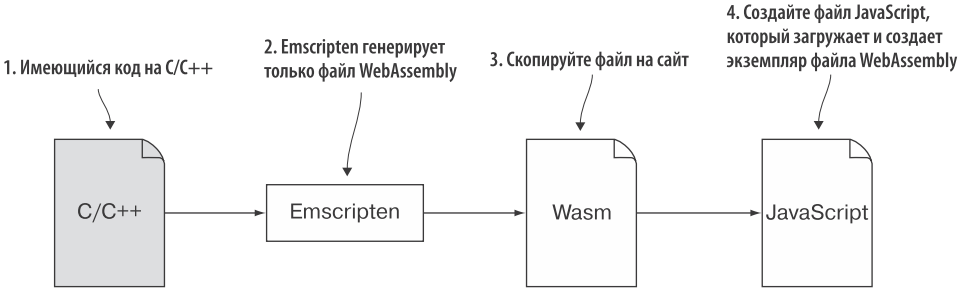


Рис. 10.9. Использование Emscripten для генерации одного лишь файла WebAssembly. Затем мы добавим JavaScript для загрузки и создания экземпляра модуля с помощью JavaScript API в WebAssembly

10.4.1. Загрузка и создание экземпляра модуля WebAssembly

Чтобы загрузить и запустить файл `side_module.wasm` из главы 3 в Node.js, вам необходимо загрузить и создать экземпляр модуля с помощью JavaScript API в WebAssembly.

Реализация серверного кода для Node.js

Первое, что вам нужно сделать, — создать папку для файлов, которые будут использоваться в этом подразделе. В папке WebAssembly\ создайте папку Chapter 10\10.4.1 SideModuleIncrement\backend\, а затем выполните следующие действия:

- скопируйте файл `side_module.wasm` из папки Chapter 3\3.5.1 side_module\ в только что созданную папку backend\;
- создайте файл `side_module_nodejs.js` в папке backend\, а затем откройте его в своем любимом редакторе.

Node.js уже запущен на сервере, поэтому вам не нужно получать файл `.wasm`, поскольку он находится на жестком диске в тех же папках, что и файлы JavaScript. Вместо этого мы будем с помощью модуля файловой системы в Node.js читать

содержимое бинарного файла WebAssembly. Затем, после того как оно будет получено, процесс вызова `WebAssembly.instantiate` и работы с модулем будет таким же, как и в браузере.

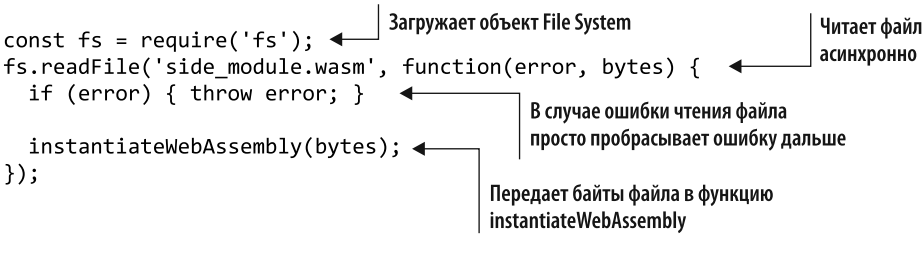
Вы включаете модуль `File System`, используя функцию `require` и передавая строку `'fs'`. Функция `require` возвращает объект, который дает доступ к различным функциям файловой системы, таким как `readFile` и `writeFile`. В этой главе мы рассмотрим только функцию `readFile`, но если вам интересно узнать больше об объекте `File System` в Node.js и доступных функциях, то посетите страницу https://nodejs.org/api/fs.html.

Будем использовать функцию `readFile` в `File System` для асинхронного чтения содержимого файла `side_module.wasm`. Функция `readFile` принимает три параметра. Первый — это путь к файлу для чтения. Второй — необязателен, он позволяет указать такие параметры, как кодировка файла. В данной главе мы не будем использовать второй параметр. Третий параметр — это функция обратного вызова, которая получит либо объект ошибки (если при чтении содержимого файла возникла проблема), либо байты файла, если чтение прошло успешно.

ИНФОБОКС

Если хотите узнать больше о функции readFile модуля File System и о втором необязательном параметре, то посетите страницу http://mng.bz/rPjy.

Добавьте следующий фрагмент кода в файл `side_module_nodejs.js`, чтобы загрузить объект `File System` (`'fs'`), а затем вызовите функцию `readFile`. Если в функцию обратного вызова передается ошибка, то в результате выбрасывается ошибка. В противном случае полученные байты передаются в функцию `instantiateWebAssembly` — ее мы и создадим далее:




```
const fs = require('fs');
fs.readFile('side_module.wasm', function(error, bytes) {
  if (error) { throw error; }

  instantiateWebAssembly(bytes);
});
```

Создайте функцию `instantiateWebAssembly`, которая принимает параметр `bytes`. Внутри нее создайте объект JavaScript `importObject` и объект `env` со значением `0` (ноль) для свойства `__memory_base`. Затем вам нужно вызвать функцию `WebAssembly.instantiate`, передав ей полученные байты, а также `importObject`. Наконец, вызовите экспортированную функцию `_Increment` из модуля WebAssembly внутри метода `then`, передав значение `2`. Выведите результат в консоль.

Функция `instantiateWebAssembly` в файле `side_module_nodejs.js` должна соответствовать коду, показанному в листинге 10.3.

Листинг 10.3. Функция instantiateWebAssembly

```
function instantiateWebAssembly(bytes) {
  const importObject = {
    env: {
      __memory_base: 0,
    }
  };

  WebAssembly.instantiate(bytes, importObject).then(result => {
    const value = result.instance.exports._Increment(2);
    console.log(value.toString());
  });
}
```

Показывает результат в окне консоли

Просмотр результатов

Чтобы запустить файл JavaScript в Node.js, откройте командную строку, перейдите к папке Chapter 10\10.4.1 SideModuleIncrement\backend\ и выполните следующую команду:

```
node side_module_nodejs.js
```

Результат вызова функции `_Increment` показан на рис. 10.10.

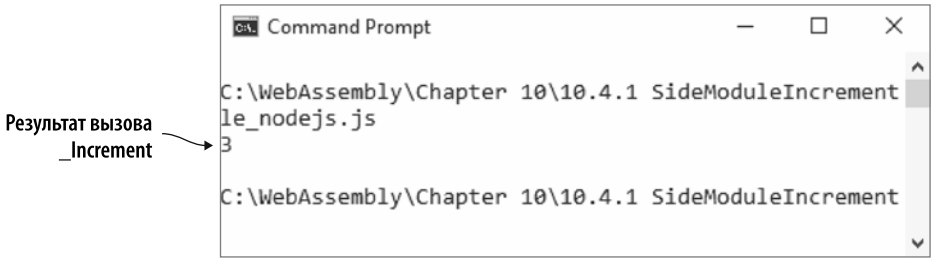


Рис. 10.10. Вывод в консоли — результат вызова функции _Increment в Node.js

10.4.2. Вызов функций в модуле WebAssembly

Последний шаг процесса, показанный на рис. 10.11, — копирование файла WebAssembly, `validate.wasm` (созданного в главе 4, в подразделе 4.2.2), в папку, в которой будут размещены файлы Node.js. Затем мы создадим файл JavaScript, который устранит пробел между взаимодействием с данными, полученными от браузера, и взаимодействием с модулем.

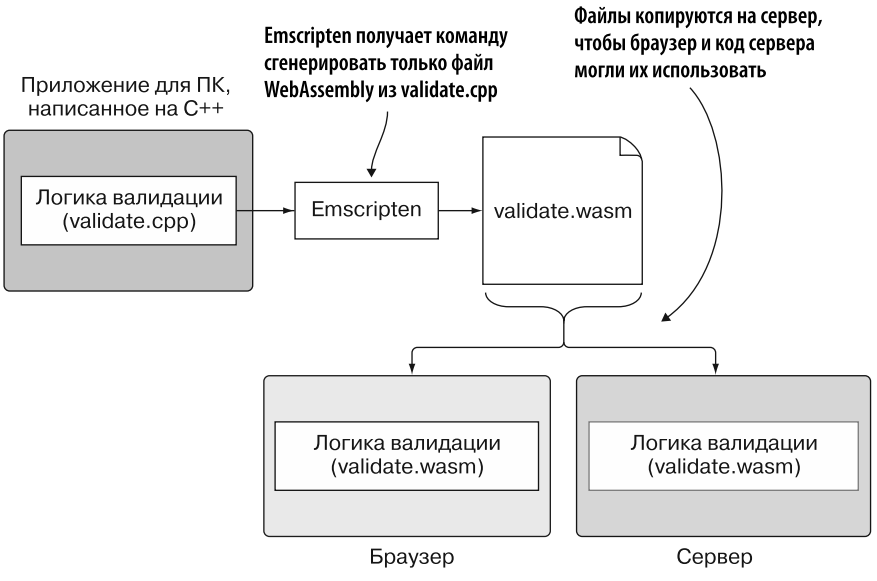


Рис. 10.11. Последний шаг процесса — копирование сгенерированного файла WebAssembly в место, в котором будут размещены файлы Node.js, и написание кода на JavaScript для взаимодействия с модулем

Реализация серверного кода для Node.js

В папке WebAssembly\ создайте папку Chapter 10\10.4.2 SideModule\backend\, а затем выполните следующие действия:

- скопируйте файлы `editproduct.js` и `validate.wasm` из папки Chapter 4\4.2 side_module\frontend\ в только что созданную папку backend\;
- переименуйте файл `editproduct.js` в `nodejs_validate.js` и откройте его в своем любимом редакторе.

JavaScript в файле `nodejs_validate.js` был написан для работы в браузере, поэтому нужно будет внести несколько изменений, чтобы он корректно работал в Node.js.

JavaScript-код использует JavaScript-объект `TextEncoder` для копирования строк в память модуля. В Node.js объект `TextEncoder` является частью пакета `util`. Первое, что вам нужно сделать в файле JavaScript, — добавить в начало файла функцию `require` для пакета `util`, как показано ниже:

```
const util = require('util');
```

Загружает пакет util для доступа к объекту TextEncoder

Далее переименуйте объект `initialData`, присвоив ему имя `clientData`:

```
const clientData = {
  name: "Women's Mid Rise Skinny Jeans",
  categoryId: "100",
};
```

Переименован из initialData

В файле `nodejs_validate.js` непосредственно перед функцией `initializePage` добавьте следующий код, чтобы байты из файла `validate.wasm` считывались и передавались в функцию `instantiateWebAssembly`:

```
const fs = require('fs');
fs.readFile('validate.wasm', function(error, bytes) {
  if (error) { throw error; }

  instantiateWebAssembly(bytes);
});
```

Читает байты файла validate.wasm

Передает байты в данную функцию

Внесите следующие изменения в функцию `initializePage`:

- переименуйте функцию в `instantiateWebAssembly` и добавьте параметр `bytes`;
- удалите строку кода, которая задает `name`, а также следующий за ней код для `category`, чтобы функция `instantiateWebAssembly` начиналась со строки кода `moduleMemory`;
- замените `WebAssembly.instantiateStreaming` на `WebAssembly.instantiate` и замените параметр `fetch ("validate.wasm")` на `bytes`;
- наконец, в методе `then` вызова `WebAssembly.instantiate` и после строки `moduleExports` добавьте вызов функции `validateData`, которую вы скоро создадите.

Измененная функция `initializePage` в файле `nodejs_validate.js` теперь должна соответствовать коду, показанному в листинге 10.4.

Листинг 10.4. Функция initializePage, переименованная в instantiateWebAssembly

```
...

function instantiateWebAssembly(bytes) {
  moduleMemory = new WebAssembly.Memory({initial: 256});

  const importObject = {
    env: {
      __memory_base: 0,
      memory: moduleMemory,
    }
  };

  WebAssembly.instantiate(bytes, importObject).then(result => {
```

Функция была переименована из initializePage, также был добавлен параметр bytes

Использует функцию instantiate вместо instantiateStreaming и передает параметр bytes вместо вызова fetch

```
    moduleExports = result.instance.exports;
    validateData();
  });
}
...
```

Вызывает validateData, после того как был создан экземпляр модуля

В файле `nodejs_validate.js` удалите функцию `getSelectedCategoryId`. Затем замените содержимое функции `setErrorMessage` вызовом `console.log` для параметра ошибки, как показано ниже:

```
function setErrorMessage(error) { console.log(error); }
```

Выводит в консоль любые сообщения об ошибках

Следующее изменение, которое необходимо внести в файл `nodejs_validate.js`, — переименовать функцию `onClickSave` в `validateData`, чтобы она вызывалась после того, как будет создан экземпляр модуля. В функции `validateData` удалите две строки кода над оператором `if`, которые получают `name` и `categoryId`. В операторе `if` перед переменными `name` и `categoryId` укажите префикс объекта `clientData`.

Функция `validateData` в файле `nodejs_validate.js` теперь должна соответствовать коду, показанному в листинге 10.5.

Листинг 10.5. Функция onClickSave, переименованная в validateData

```
...

function validateData() {
  let errorMessage = "";
  const errorMessagePointer = moduleExports._create_buffer(256);

  if (!validateName(clientData.name, errorMessagePointer) ||
      !validateCategory(clientData.categoryId,
      errorMessagePointer)) {
    errorMessage = getStringFromMemory(errorMessagePointer);
  }
  moduleExports._free_buffer(errorMessagePointer);

  setErrorMessage(errorMessage);
  if (errorMessage === "") {
  }
}
...
```

Переименована из onClickSave

Значение name объекта clientData передается в validateName

Значение categoryId объекта clientData передается в validateName

Проблем при валидации не обнаружено. Данные можно сохранить

Последняя область, которую нужно изменить, — функция `copyStringToMemory`. В браузере объект `TextEncoder` является глобальным; но в Node.js объект находится

в пакете `util`. В файле `nodejs_validate.js` вам необходимо добавить префикс объекта `TextEncoder` к загруженному ранее объекту `util`, как показано ниже:

```
function copyStringToMemory(value, memoryOffset) {
  const bytes = new Uint8Array(moduleMemory.buffer);
  bytes.set(new util.TextEncoder().encode((value + "\0")),
      memoryOffset);
}
```

Объект TextEncoder в Node.js входит в пакет util

Никаких других изменений в JavaScript-коде файла `nodejs_validate.js` не требуется.

Просмотр результатов

Чтобы проверить логику, подправьте данные, изменив значение свойства `categoryId` на значение, которого нет в массиве `VALID_CATEGORY_IDS` (`categoryId: "1001"`). Для запуска файла JavaScript в Node.js откройте командную строку, перейдите в папку Chapter 10\10.4.2 SideModule\backend\ и выполните следующую команду:

```
node nodejs_validate.js
```

Должно появиться сообщение об ошибке валидации, показанное на рис. 10.12.

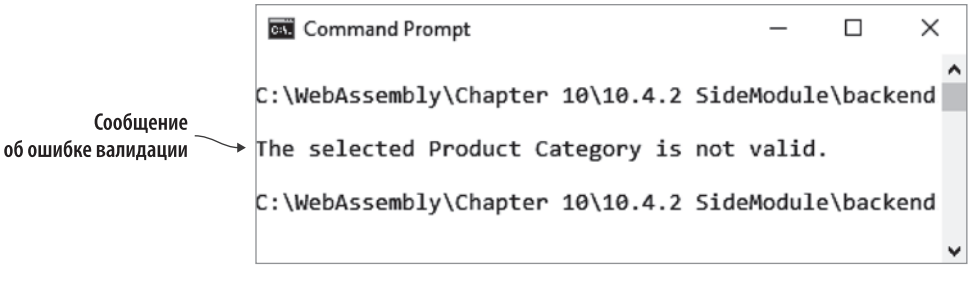


Рис. 10.12. Ошибка валидации категории товара в Node.js

В этом подразделе было описано, как изменить JavaScript-код для загрузки и создания экземпляра модуля WebAssembly, вызываемого вашим кодом. В следующем подразделе вы узнаете, как работать с модулем, вызывающим ваш JavaScript-код.

10.4.3. Вызов JavaScript из модуля WebAssembly

Например, вызывающий JavaScript напрямую модуль пригодится, если ему нужно выполнить длительную операцию. Вместо того чтобы JavaScript выполнял вызов функции и ждал результатов, модуль мог бы периодически вызывать JavaScript

в целях получения дополнительной информации или предоставления обновлений о своем состоянии.

Если вы не используете сгенерированный Emscripten JavaScript (что и будет показано в этом подразделе), то все должно быть немного иначе, поскольку весь код JavaScript находится в одной области. В результате модуль может вызывать JavaScript и иметь доступ к основному коду, как показано на рис. 10.13.

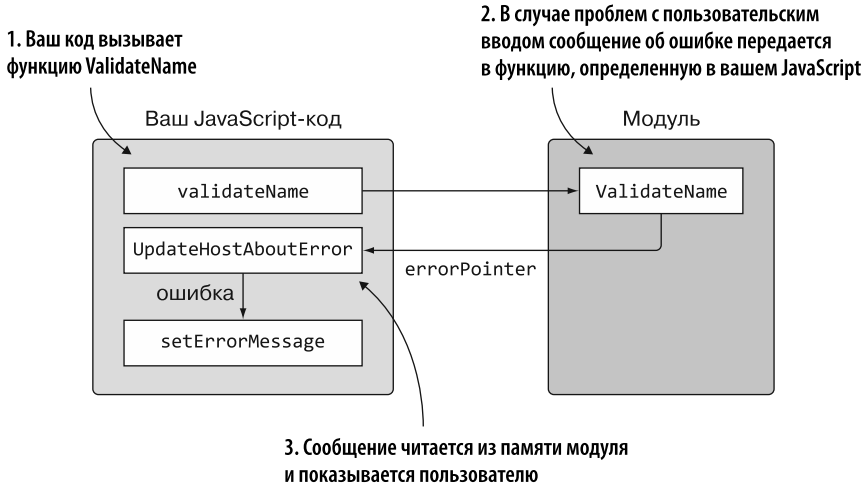


Рис. 10.13. Как будет работать логика обратного вызова, если сгенерированный Emscripten JavaScript не используется

Реализация серверного кода для Node.js

В папке WebAssembly\ создайте папку Chapter 10\10.4.3 SideModuleCallingJS\backend\, а затем выполните следующие действия:

- скопируйте файлы `editproduct.js` и `validate.wasm` из папки Chapter 5\5.2.1 SideModuleCallingJS\frontend\ в только что созданную папку backend\;
- переименуйте файл `editproduct.js` в `nodejs_validate.js`, а затем откройте его в своем любимом редакторе.

Файл `nodejs_validate.js` нужно изменить для работы с Node.js. Код использует JavaScript-объект `TextEncoder` в функции `copyStringToMemory`; в Node.js объект `TextEncoder` является частью пакета `util`. Нужно будет включить ссылку на пакет, чтобы код мог использовать данный объект. Добавьте следующую строку в начало файла `nodejs_validate.js`:

```
const util = require('util');
```

Загружает пакет util, чтобы получить доступ к объекту TextEncoder

Переименуйте объект `initialData` в `clientData`. Затем в файле `nodejs_validate.js` перед функцией `initializePage` добавьте код, показанный в следующем фрагменте, чтобы прочитать байты из файла `validate.wasm` и передать их в функцию `instantiateWebAssembly`:

```
const fs = require('fs');
fs.readFile('validate.wasm', function(error, bytes) {
  if (error) { throw error; }

  instantiateWebAssembly(bytes);
});
```

Читает байты файла validate.wasm

Передает байты в данную функцию

Затем вам необходимо изменить функцию `initializePage`, выполнив следующие действия:

- переименуйте функцию в `instantiateWebAssembly` и добавьте параметр `bytes`;
- удалите строки кода перед строкой `moduleMemory`;
- измените `WebAssembly.instantiateStreaming` на `WebAssembly.instantiate` и замените значение параметра `fetch ("validate.wasm")` на `bytes`;
- добавьте вызов функции `validateData` после строки `moduleExports` в методе `then` вызова `WebAssembly.instantiate`.

Измененная функция `initializePage` в файле `nodejs_validate.js` теперь должна соответствовать коду, показанному в листинге 10.6.

Листинг 10.6. Функция initializePage, переименованная в instantiateWebAssembly

```
...

function instantiateWebAssembly(bytes) {
  moduleMemory = new WebAssembly.Memory({initial: 256});

  const importObject = {
    env: {
      __memory_base: 0,
      memory: moduleMemory,
      _UpdateHostAboutError: function(errorMessagePointer) {
        setErrorMessage(getStringFromMemory(errorMessagePointer));
      },
    }
  };

  WebAssembly.instantiate(bytes, importObject).then(result => {
    moduleExports = result.instance.exports;
    validateData();
  });
}

...
```

Функция была переименована из initializePage, также был добавлен параметр bytes

Использует функцию instantiate вместо instantiateStreaming и передает параметр bytes вместо вызова fetch

Вызывает validateData, после того как был создан экземпляр модуля

В файле `nodejs_validate.js` удалите функцию `getSelectedCategoryId`. Затем замените содержимое функции `setErrorMessage` вызовом `console.log` для параметра ошибки, как показано ниже:

```
function setErrorMessage(error) { console.log(error); }
```

Выводит в консоль любые сообщения об ошибках

Измените функцию `onClickSave`, выполнив следующие шаги:

- переименуйте функцию в `validateData`;
- удалите строки кода `setErrorMessage()`, `const name` и `const categoryId`;
- добавьте префикс объекта `clientData` к значениям `name` и `categoryId` в операторах `if`.

Измененная функция `onClickSave` в файле `nodejs_validate.js` теперь должна выглядеть так:

Переименована из onClickSave

```
function validateData() {
    if (validateName(clientData.name) &&
        validateCategory(clientData.categoryId)) {
    }
}
```

Значение свойства name объекта clientData передается в validateName

Значение свойства categoryId объекта clientData передается в validateCategory

Проблем при валидации не обнаружено. Данные могут быть сохранены

Последнее, что нам нужно изменить, — функция `copyStringToMemory`. Необходимо добавить префикс объекта `TextEncoder` к загруженному ранее объекту `util`.

Функция `copyStringToMemory` в файле `nodejs_validate.js` должна соответствовать коду, показанному ниже:

```
function copyStringToMemory(value, memoryOffset) {
  const bytes = new Uint8Array(moduleMemory.buffer);
  bytes.set(new util.TextEncoder().encode((value + "\0")),
      memoryOffset);
}
```

Объект TextEncoder в Node.js входит в пакет util

Никаких других изменений в файле `nodejs_validate.js` не требуется.

Просмотр результатов

Чтобы протестировать логику валидации, вы можете подправить данные в `clientData`, поменяв значение свойства `name` на значение, превышающее `MAXIMUM_NAME_LENGTH`, равное 50 символам (`name: "This is a very long product name to test the validation logic."`).

Откройте командную строку, перейдите в папку Chapter 10\10.4.3 SideModuleCallingJS\backend\ и выполните следующую команду:

```
node nodejs_validate.js
```

Сообщение показано на рис. 10.14.

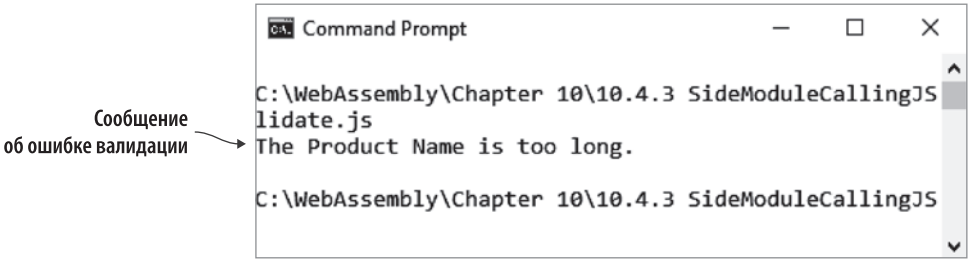


Рис. 10.14. Сообщение об ошибке валидации длины названия товара в Node.js

В этом подразделе было описано, как загружать модуль WebAssembly, который напрямую вызывает ваш код JavaScript, и как работать с ним. В следующем подразделе вы узнаете, как использовать модуль, вызывающий указатели на функции JavaScript.

10.4.4. Вызов указателей на функции JavaScript из модуля WebAssembly

Возможность передать модулю указатель на функцию JavaScript добавляет гибкости вашему коду по сравнению с прямым вызовом JavaScript, поскольку вы не зависите от какой-то одной конкретной функции. Вместо этого модулю может быть передана функция по запросу, если ее сигнатура соответствует ожидаемой.

Кроме того, в зависимости от того, как настроен JavaScript, вызов нужной функции в вашем JavaScript-коде может потребовать нескольких дополнительных вызовов. С помощью указателя на функцию модуль вызывает функцию напрямую.

Модули WebAssembly могут задействовать указатели, которые указывают на функции внутри модуля или импортированные функции. В этом случае мы будем применять модуль WebAssembly, созданный в разделе 6.2 главы 6, предполагающий, что будут заданы функции `OnSuccess` и `OnError`, как показано на рис. 10.15. При использовании любой функции модуль вызывает именно JavaScript-код.

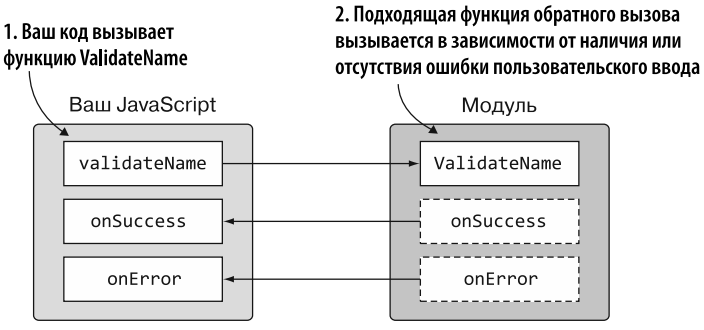


Рис. 10.15. Модуль, импортирующий JavaScript-функции onSuccess и onError во время создания экземпляра. В случае их использования функция ValidateName модуля вызывает именно JavaScript-код

Реализация серверного кода для Node.js

Теперь нам нужно изменить JavaScript-код, написанный для использования в браузере в главе 6, чтобы он мог работать в Node.js. В папке WebAssembly\ создайте папку Chapter 10\10.4.4 SideModuleFunctionPointers\backend\, а затем выполните следующие действия:

- скопируйте файлы `editproduct.js` и `validate.wasm` из папки Chapter 6\6.2.2 SideModuleFunctionPointers\frontend\ в только что созданную папку backend\;
- переименуйте файл `editproduct.js` в `nodejs_validate.js`, а затем откройте его в своем любимом редакторе.

В вашем JavaScript-коде используется JavaScript-объект `TextEncoder`. Поскольку объект является частью пакета `util` в Node.js, первое, что вам нужно сделать, — включить ссылку на пакет. Добавьте код, показанный ниже, в начало файла `nodejs_validate.js`:

```
const util = require('util');
```

Загружает пакет util, чтобы получить доступ к объекту TextEncoder

Переименуйте объект `initialData` в `clientData`.

В файле `nodejs_validate.js` перед функцией `initializePage` добавьте следующий код для чтения байтов из файла `validate.wasm` и передачи их в функцию `instantiateWebAssembly`:

```
const fs = require('fs');
fs.readFile('validate.wasm', function(error, bytes) {
  if (error) { throw error; }

  instantiateWebAssembly(bytes);
});
```

Читает байты файла validate.wasm

Передает байты в данную функцию

Измените функцию `initializePage`, выполнив следующие действия:

- переименуйте функцию в `instantiateWebAssembly` и добавьте параметр `bytes`;
- удалите строки кода перед строкой `moduleMemory`;
- измените `WebAssembly.instantiateStreaming` на `WebAssembly.instantiate` и замените значение параметра `fetch ("validate.wasm")` на `bytes`;
- добавьте вызов функции `validateData` в метод `then` в `WebAssembly.instantiate` после последнего вызова функции `addToTable`.

Измененная функция `initializePage` в файле `nodejs_validate.js` теперь должна соответствовать коду, показанному в листинге 10.7.

Листинг 10.7. Функция initializePage, переименованная в instantiateWebAssembly

```
...
function instantiateWebAssembly(bytes) {
  moduleMemory = new WebAssembly.Memory({initial: 256});

  moduleTable = new WebAssembly.Table({initial: 1, element: "anyfunc"});
    const importObject = {
      env: {
        __memory_base: 0,
        memory: moduleMemory,
        __table_base: 0,
        table: moduleTable,
        abort: function(i) { throw new Error('abort'); },
      }
    };

    WebAssembly.instantiate(bytes, importObject).then(result => {
      moduleExports = result.instance.exports;
    validateOnSuccessNameIndex = addToTable(() => {
        onSuccessCallback(validateNameCallbacks);
      }, 'v');

    validateOnSuccessCategoryIndex = addToTable(() => {
      onSuccessCallback(validateCategoryCallbacks);
    }, 'v');

    validateOnErrorNameIndex = addToTable((errorMessagePointer) => {
      onErrorCallback(validateNameCallbacks, errorMessagePointer);
    }, 'vi');

    validateOnErrorCategoryIndex = addToTable((errorMessagePointer) => {
      onErrorCallback(validateCategoryCallbacks, errorMessagePointer);
  }, 'vi'); validateData();
  });
}
...
```

Функция была переименована из initializePage, также был добавлен параметр bytes

Использует функцию instantiate вместо instantiateStreaming и передает параметр bytes вместо вызова fetch

Вызывает validateData, после того как был создан экземпляр модуля

Следующее изменение, которое нужно внести в файл `nodejs_validate.js`, — удалить функцию `getSelectedCategoryId`. Затем нужно заменить содержимое функции `setErrorMessage` вызовом `console.log` для параметра `error`:

```
function setErrorMessage(error) { console.log(error); }
```

Выводит в консоль любые сообщения об ошибках

Измените функцию `onClickSave`, выполнив следующие шаги:

- переименуйте функцию в `validateData`;
- удалите строки кода `setErrorMessage()`, `const name` и `const categoryId`;
- добавьте префикс объекта `clientData` к значениям `name` и `categoryId`, которые передаются в функции `validateName` и `validateCategory`.

Измененная функция `onClickSave` в файле `nodejs_validate.js` теперь должна соответствовать коду, показанному в листинге 10.8.

Листинг 10.8. Функция onClickSave, переименованная в validateData

```
...

function validateData() {
  Promise.all([
    validateName(clientData.name),
    validateCategory(clientData.categoryId)
  ])
  .then(() => {

  })
  .catch((error) => {
    setErrorMessage(error);
  });
}
...
```

Переименована из onClickSave

Значение свойства name объекта clientData передается в validateName

Значение свойства categoryId объекта clientData передается в validateCategory

Проблем при валидации не обнаружено. Данные можно сохранить

Последнее, что нам нужно изменить, — функция `copyStringToMemory`. Необходимо добавить префикс объекта `TextEncoder` к загруженному ранее объекту `util`. Функция `copyStringToMemory` в файле `nodejs_validate.js` должна соответствовать коду, показанному в следующем фрагменте:

```
function copyStringToMemory(value, memoryOffset) {
  const bytes = new Uint8Array(moduleMemory.buffer);
  bytes.set(new util.TextEncoder().encode((value + "\0")),
      memoryOffset);
}
```

Объект TextEncoder в Node.js входит в пакет util

Никаких других изменений в файле `nodejs_validate.js` не требуется.

Просмотр результатов

Чтобы протестировать логику валидации, вы можете подправить данные в `clientData`, удалив значение свойства `name` (`name: ""`) и сохранив файл. Откройте командную строку, перейдите в папку Chapter 10\10.4.4 SideModuleFunctionPointers\backend\ и выполните следующую команду:

```
node nodejs_validate.js
```

Вы должны увидеть сообщение, показанное на рис. 10.16.

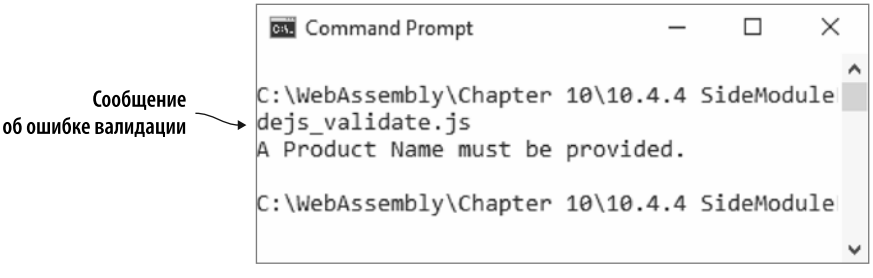


Рис. 10.16. Ошибка валидации названия товара в Node.js

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже приведены некоторые возможные варианты использования того, что вы узнали в этой главе.

- Как вы увидели в данной главе, Node.js можно запускать из командной строки, что означает следующее: логику WebAssembly разработчик может использовать локально на своей машине — это упростит повседневные задачи.
- С помощью веб-сокетов в Node.js можно реализовать совместную работу в реальном времени в вашем веб-приложении.
- Вы можете использовать Node.js, чтобы добавить в игру возможность обмена текстовыми сообщениями между участниками.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какую функцию Node.js необходимо вызвать для загрузки файла JavaScript, созданного Emscripten?
2. Какое свойство объекта `Module` в Emscripten нужно реализовать, чтобы получать информацию о том, когда модуль WebAssembly готов к взаимодействию?
3. Как бы вы изменили файл `index.js` из главы 8, чтобы логика динамического связывания работала в Node.js?

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Модули WebAssembly можно применять в Node.js, а необходимый JavaScript-код очень похож на тот, который вы использовали при работе в браузере.
- Модули, включающие код JavaScript в Emscripten, могут загружаться и создавать экземпляры при загрузке JavaScript с помощью функции `require`. Однако, в отличие от браузера, глобальные вспомогательные функции Emscripten недоступны. Доступ ко всем функциям в файле JavaScript, созданном Emscripten, должен осуществляться через возвращаемый объект функции `require`.
- Node.js не поддерживает функцию `WebAssembly.instantiateStreaming`. Вместо этого нужно использовать функцию `WebAssembly.instantiate`. Если вы написали один файл JavaScript в целях применения модуля WebAssembly как в браузере, так и в Node.js, то вам понадобится функция распознавания окружения, о которой вы узнали в главе 3, в разделе 3.6.
- При загрузке файла WebAssembly вручную в Node.js метод `fetch` не используется, поскольку файл WebAssembly находится на том же компьютере, что и исполняемый код JavaScript. Вместо этого вы читаете байты файла WebAssembly из `File System`, а затем передаете их в функцию `WebAssembly.instantiate`.
- Из-за проблем с областью действия между кодом, вызывающим функцию `require`, и JavaScript, сгенерированным Emscripten, пользовательский JavaScript-код в JavaScript-файле Emscripten должен быть автономным и не должен вызывать родительский код.

Часть IV

Отладка и тестирование

В большинстве случаев при разработке наступает время, когда мы сталкиваемся с проблемой, которую необходимо отследить. Эта задача может быть не сложнее, чем чтение кода, но иногда приходится копать глубже. В данной части книги вы узнаете о вариантах, доступных для отладки и тестирования модуля WebAssembly.

В главе 11 вы узнаете о текстовом формате WebAssembly, создав игру со сравнением карт. В главе 12 расширите ее, чтобы узнать о различных параметрах, доступных для отладки модуля WebAssembly. А глава 13 подведет итоги изучения навыков разработки WebAssembly, научив вас писать интеграционные тесты для модулей.

11 Текстовый формат WebAssembly

В этой главе

- ✓ Создание версии модуля WebAssembly в текстовом формате.
- ✓ Компиляция кода текстового формата в двоичный модуль с помощью онлайн-инструмента WebAssembly Binary Toolkit.
- ✓ Связывание модуля, сгенерированного с применением Binary Toolkit, с модулем, сгенерированным Emscripten.
- ✓ Создание HTML и JavaScript для пользовательского интерфейса игры.

WebAssembly разработан с использованием двоичного формата файла, поэтому его файлы имеют минимальный размер, что обеспечивает быструю передачу и загрузку; это не означает, что разработчики могут скрывать свой код. В действительности все как раз наоборот. WebAssembly разработан с учетом открытости Интернета. В результате также доступен текстовый формат, эквивалент двоичного.

Текстовый формат позволяет пользователям браузера проверять WebAssembly-код веб-страницы почти так же, как они проверяют JavaScript. Вдобавок эквивалент двоичного формата, текстовый, предоставляется для отладки в браузере, если модуль WebAssembly не включает карты исходников, как показано на рис. 11.1.

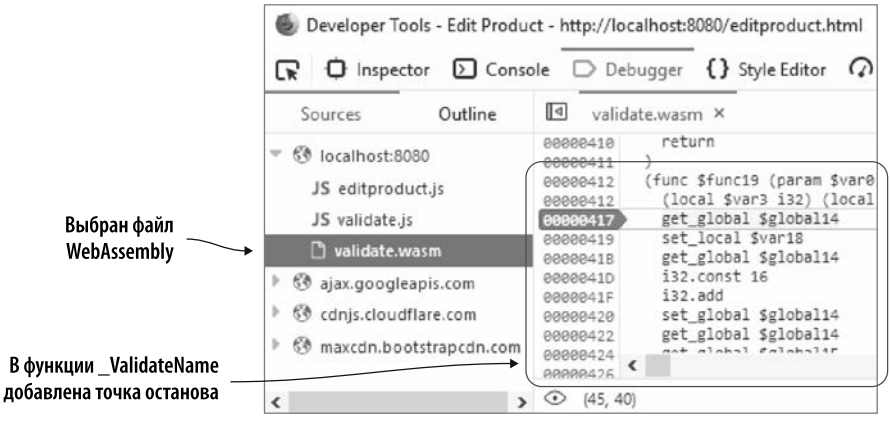


Рис. 11.1. Инструменты разработчика в Firefox с точкой останова в функции _ValidateName модуля, созданного в главе 4, в разделе 4.1

Предположим, вам нужно создать игру со сравнением карт, показанную на рис. 11.2. Уровень 1 начнется с двух рядов по две карты, все лицом вниз. Игрок щелкает кнопкой мыши на двух картах, и при нажатии они поворачиваются лицом вверх. Одинаковые карты исчезают. Если карты не совпадают, то снова переворачиваются рубашкой вверх.

Рис. 11.2. Уровень 1 игры со сравнением карт — одновременно открыты две неодинаковые карты, поэтому они повернутся лицом вниз

Игрок выиграет уровень, если все карты исчезнут. Как показано на рис. 11.3, в случае выигрыша игра показывает сообщение, которое дает возможность повторно пройти текущий уровень или перейти на следующий.

Окно результатов при победе игрока. Ему предоставляется возможность снова сыграть на текущем уровне или перейти на следующий

Рис. 11.3. Когда игрок выигрывает, ему предлагают снова сыграть на текущем уровне или перейти на следующий

Мы рассмотрим отладку модуля WebAssembly в следующей главе, но прежде необходимо иметь представление о текстовом формате и о том, как он работает. В данной главе мы создадим основную логику карточной игры, используя текстовый формат WebAssembly, чтобы увидеть в деталях, как это работает. Затем скомпилируем его в модуль WebAssembly с помощью онлайн-инструмента WebAssembly Binary Toolkit. HTML, CSS и изображения будут применяться для пользовательского интерфейса игры.

Создавая модуль с применением только текстового формата, вы не будете иметь доступа к стандартным функциям библиотеки C, таким как `malloc` и `free`. В качестве обходного пути создадим простой модуль с помощью Emscripten, который будет экспортировать дополнительные функции, необходимые модулю текстового формата.

На рис. 11.4 показан порядок создания игры из этой главы.

1. Добавьте основную логику игры, используя текстовый формат WebAssembly.
2. Задействуйте набор инструментов WebAssembly Binary Toolkit, чтобы создать модуль WebAssembly из текстового формата (`card.wasm`).
3. Создайте файл на C++, который позволит модулю `cards.wasm` получить доступ к некоторым стандартным функциям библиотеки C.
4. Используйте Emscripten, чтобы создать модуль WebAssembly из файла на C++.

5. Скопируйте сгенерированные файлы WebAssembly на сервер, чтобы браузер мог их использовать. Затем создайте HTML и JavaScript, которые загрузят и свяжут два модуля WebAssembly. Добавьте также JavaScript, который будет передавать в модуль информацию о действиях игрока.

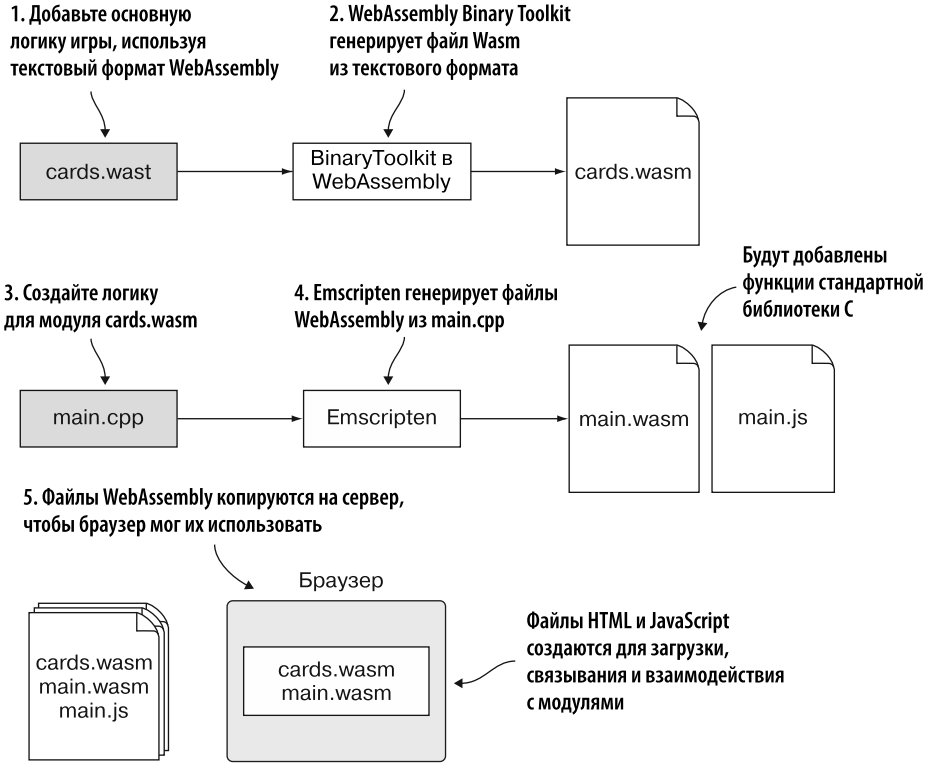


Рис. 11.4. Шаги по созданию игры

11.1. СОЗДАНИЕ ОСНОВНОЙ ЛОГИКИ ИГРЫ С ПОМОЩЬЮ ТЕКСТОВОГО ФОРМАТА WEBASSEMBLY

В текстовом формате WebAssembly используются *узлы s-выражений*, которые позволяют легко представить элементы модуля.

В текстовом формате WebAssembly каждое s-выражение заключено в круглые скобки, а первый элемент в скобках — это метка, указывающая тип узла. После метки узел может иметь список атрибутов, разделенных пробелами, или даже другие узлы. Поскольку текстовый формат предназначен для чтения людьми,

дочерние узлы обычно разделяются переводом строки и имеют отступ, чтобы показать отношения «родитель/потомок».

НАПОМИНАНИЕ

S-выражение (сокращение от символьного (symbolic) выражения) впервые появилось в языке программирования Lisp. Оно может быть атомарным или составленным из упорядоченных пар s-выражений, что позволяет вкладывать s-выражения друг в друга. Атом — это символ, не являющийся списком: например, foo или 23. Список обозначается круглыми скобками и может быть пустым или содержать атомарные выражения или даже другие списки. Элементы разделяются пробелами, например (), или (foo), или (foo (bar 132)).

В текстовом формате вы можете ссылаться на большинство элементов, например на функцию или параметр, по индексу элемента. Однако при наличии нескольких функций или переменных ссылка на все по индексу иногда может сбивать с толку. Можно дополнительно указать имя переменной для элемента при его определении, что мы и сделаем для всех переменных и функций в этой главе.

Имена переменных в текстовом формате начинаются с символа `$`, за которым следуют буквенно-цифровые символы, указывающие, что представляет собой переменная. Обычно ее имя представляет тип данных, для которого она предназначена, например `$func` для функции, но также можно использовать имя переменной, например `$add` для функции сложения. Иногда даже можно увидеть, что имя переменной заканчивается числом, обозначающим ее индекс, например `$func0`.

WebAssembly поддерживает четыре типа значений (32- и 64-битные целые числа, 32- и 64-битные числа с плавающей запятой). Логические значения представлены 32-битным целым числом. Все другие типы значений, такие как строки, должны быть представлены в линейной памяти модуля. В текстовом формате представлены четыре типа значений:

- `i32` — для 32-битного целого числа;
- `i64` — для 64-битного целого числа;
- `f32` — для 32-битного числа с плавающей запятой;
- `f64` — для 64-битного числа с плавающей запятой.

Чтобы упростить работу с четырьмя типами данных, в текстовом формате добавлен объект для каждого типа с именем этого типа. Например, чтобы сложить два значения `i32`, нужно использовать `i32.add`. Еще один пример: если вам нужно использовать значение с плавающей запятой `10.5`, примените `f32.const 10.5`. Список команд памяти и числовых инструкций для типов объектов доступен на сайте http://webassembly.github.io/spec/core/text/instructions.html.

11.1.1. Разделы модуля

В главе 2 вы узнали об известных и пользовательских разделах модуля. Каждый из известных разделов имеет конкретное назначение, четко определен и валидируется при создании экземпляра модуля WebAssembly. Пользовательские разделы служат для данных, которые не относятся к известным, и не вызовут ошибку валидации, если данные размещены неправильно.

На рис. 11.5 представлена базовая структура двоичного байт-кода. Каждый известный раздел необязателен, но если добавляется, то его можно указать только один раз. Пользовательские разделы также не являются обязательными, но, будучи в наличии, могут быть размещены до известных разделов, после них или между ними.

Как показано в табл. 11.1, текстовый формат использует метки s-выражений, которые соответствуют известным разделам двоичного формата.

Таблица 11.1. Известные разделы и соответствующие им метки s-выражений

Двоичный формат	Текстовый формат	Двоичный формат	Текстовый формат
Преамбула	module	Глобальные переменные	global
Тип	type	Экспорт	export
Импорт	import	Старт	start
Функция	func	Элемент	elem
Таблица	table	Код	
Память	memory	Данные	data

Из таблицы вы могли заметить, что текстовый формат, эквивалентный разделу «Код» двоичного формата, не указан. В двоичном формате сигнатура функции и тело функции находятся в отдельных разделах. В текстовом формате тело функции включается в функцию как часть s-выражения `func`.

В двоичном формате каждый известный раздел необязателен, но его можно добавить только один раз и он должен появляться в порядке, указанном в табл. 11.1. С другой стороны, в текстовом формате единственный узел, положение которого имеет значение, — это s-выражение `import`. Если оно добавлено, то должно располагаться перед s-выражениями `table`, `memory`, `global` и `func`.

СОВЕТ

Для удобства сопровождения кода рекомендуется хранить все связанные узлы вместе, а разделы располагать в том же порядке, в каком вы ожидаете увидеть эти разделы в двоичном файле.

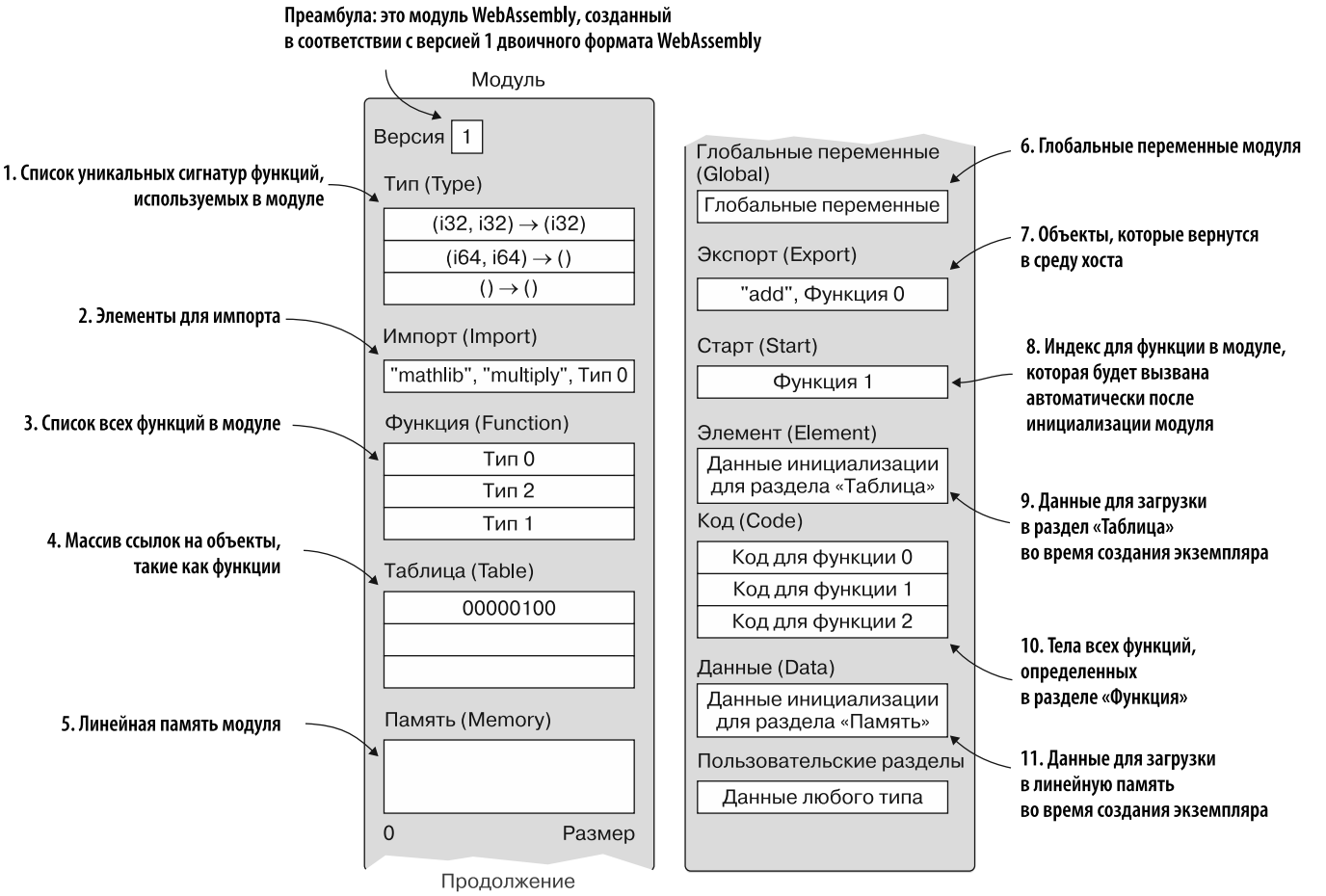


Рис. 11.5. Базовая структура двоичного байт-кода с примерами известных и пользовательских разделов

11.1.2. Комментарии

Существует два способа добавить комментарий в код текстового формата. Двойная точка с запятой используется для однострочного комментария, и все, что находится справа от точки с запятой, закомментировано, как в следующем примере:

```
;; this is a single-line comment
```

Если вы хотите закомментировать часть кода — либо часть элемента, либо сразу несколько элементов, — то начните комментарий с открывающей круглой скобки и точки с запятой, а затем закройте комментарий точкой с запятой и закрывающей скобкой. Некоторые инструменты включают эти типы комментариев в элементы, чтобы указать, чему именно соответствует какой-либо индекс, как в следующем примере:

```
(; 0 ;)
```

В ряд известных разделов, которые вы определите для игры, вам нужно будет включить сигнатуру функции. Сигнатуры функций используются сразу в нескольких разделах, поэтому ниже будет рассказано о них.

11.1.3. Сигнатуры функций

Сигнатура функции — это определение функции без тела. S-выражение для сигнатуры функции начинается с метки, в которой используется слово `func`, за которым может следовать имя переменной.

Если функция имеет параметры, то добавляется s-выражение параметра, которое указывает тип значения параметра. Например, следующая сигнатура функции имеет единственный 32-битный целочисленный параметр и не возвращает значения:

```
(func (param i32))
```

Если функция имеет несколько параметров, то можно добавить дополнительный узел `param` для каждого параметра. Например, следующая сигнатура — для функции с двумя параметрами `i32`:

```
(func (param i32) (param i32))
```

Можно также определить параметры с помощью сокращенного метода, который использует один узел `param`, но разделенный пробелами список каждого типа параметра, как в следующем примере, эквивалентном показанному ранее примеру с двумя узлами `param`:

```
(func (param i32 i32))
```

Если функция имеет возвращаемое значение, то добавляется s-выражение `result`, указывающее тип этого значения. Ниже приведен пример сигнатуры, которая имеет два 32-битных параметра и возвращает 32-битное значение:

```
(func (param i32 i32) (result i32))
```

Если функция не имеет параметров или возвращаемого значения, то узлы `param` или `result` не добавляются:

```
(func)
```

Изучив некоторые основы текстового формата, можно перейти к созданию логики игры (рис. 11.6).

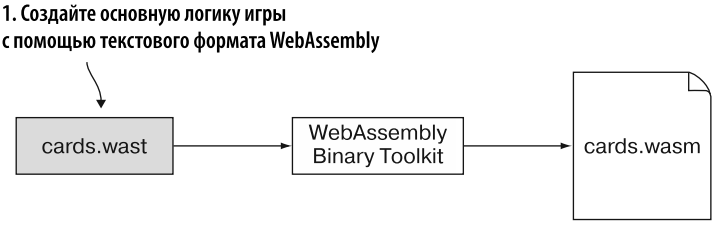


Рис. 11.6. Создание основной логики игры с помощью текстового формата WebAssembly

11.1.4. Узел module

В папке WebAssembly\ создайте папку Chapter 11\source\ для файлов, которые будут использоваться в этом подразделе. Создайте файл `cards.wast` для кода текстового формата, а затем откройте его в своем любимом редакторе.

Корневой узел s-выражения, используемый для текстового формата WebAssembly, — `module`, и все элементы модуля представлены как дочерние узлы данного узла. Все разделы модуля являются необязательными, поэтому модуль может быть пустым — он будет представлен в текстовом формате как `(module)`.

Как показано на рис. 11.7, узел `module` эквивалентен разделу преамбулы двоичного формата. Версия используемого двоичного формата будет включена в инструмент, с помощью которого текстовый формат преобразуется в файл двоичного формата.

Первый шаг при построении базовой логики игры — добавить узел `module` в файл `cards.wast`, как показано ниже:

```
(module    <-- Корневой узел module
)          <-- Все элементы модуля будут дочерними
               по отношению к узлу module
```

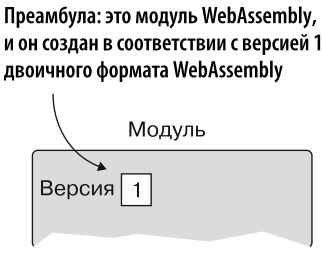


Рис. 11.7. Узел module — эквивалент раздела преамбулы в двоичном формате. Его версия будет определена инструментом, используемым для создания файла в двоичном формате

Создав узел `module`, вы можете продолжить и добавить известные разделы в качестве его дочерних узлов. Узлы `type` будут отображаться как его первые дочерние элементы, но нельзя определить, какие сигнатуры функций нужны вашему модулю, пока вы не импортируете или не создадите необходимые функции для его логики. По этой причине мы пока пропустим узлы `type`, но вернемся и добавим их, как только функции модуля будут готовы.

Первый раздел, который нужно добавить к узлу `module`, — узлы `import`.

11.1.5. Узлы import

В известном разделе «Импорт» (рис. 11.8) объявляются все элементы, которые должны быть импортированы в модуль, включая импорт функций, таблиц, памяти или глобальных переменных. Для созданного здесь модуля вы импортируете необходимую память, а также несколько функций.

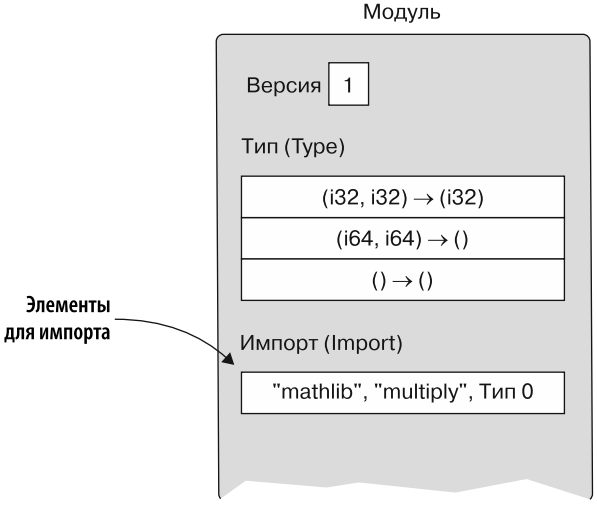


Рис. 11.8. Известный раздел «Импорт» объявляет все элементы, которые должны быть импортированы в модуль

Импорт определяется с помощью s-выражения с меткой `import`, за которой следует имя пространства имен, за ним — имя элемента, который будет импортирован, а затем — s-выражение, представляющее импортируемые данные. Чтобы соответствовать модулям, созданным Emscripten, используемое пространство имен будет называться `"env"`. Emscripten помещает символ подчеркивания перед именем импортируемого элемента, поэтому здесь мы сделаем то же самое, чтобы код JavaScript был последовательным.

Ниже приведен пример узла `import`, определенного для функции с двумя параметрами `i32` и возвращаемым значением `i32`:

```
(import "env" "_Add"
  (func $add (param i32 i32) (result i32))
)
```

"env" — название пространства имен. "_Add" — название импортируемого элемента

Это импорт для функции с двумя параметрами i32 и возвращаемым значением i32

При создании экземпляра модуля WebAssembly объект JavaScript необходимо передать функции `WebAssembly.instantiateStreaming`, которая обеспечивает импорт, ожидаемый модулем. Ниже приведен пример объекта JavaScript для модуля, ожидающего функцию `_Add`, определенную ранее:

```
const importObject = {
  env: {
    _Add: function(value1, value2) {
      return value1 + value2;
    }
  }
};
```

Название объекта должно соответствовать названию пространства имен (в нашем случае — env)

Название элемента находится слева от двоеточия, в то время как импортируемый элемент — справа

Получив представление о том, как определяются узлы `import`, перейдем к добавлению их в игру.

Добавление узлов import в игру

Логика игры должна будет импортировать некоторые функции из JavaScript, чтобы модуль мог вызывать JavaScript и обновлять его на различных этапах игры. Функции, перечисленные в табл. 11.2, будут импортированы из кода JavaScript.

Позже в этой главе с помощью Emscripten мы создадим модуль, который будет вручную связан с модулем, созданным выше, во время выполнения. Модуль, сгенерированный Emscripten, предоставит доступ к таким функциям, как `malloc` и `free`, чтобы помочь с управлением памятью. Модуль также предоставит функции, помогающие генерировать случайные числа.

Таблица 11.2. Функции JavaScript, которые нужно будет импортировать

Название элемента	Параметры	Назначение
_GenerateCards	rows, columns, level	Сообщает JavaScript, сколько строк и столбцов карт нужно создать. Параметр level предназначен для отображения уровня, на котором играет пользователь
_FlipCard	row, column, cardValue	Дает JavaScript указание перевернуть карту по указанному индексу строки и столбца. Значение cardValue, равное –1, указывает на то, что карту нужно перевернуть лицевой стороной вниз (карты не совпадают). В противном случае карта переворачивается лицом вверх, поскольку игрок только что щелкнул на ней
_RemoveCards	row1, column1, row2, column2	Дает JavaScript указание удалить две карты на основе их индексов строк и столбцов, так как они совпадают
_LevelComplete	level, anotherLevel	Сообщает JavaScript, что игрок завершил уровень, и указывает, доступен ли новый уровень. JavaScript покажет экран итогов и позволит игроку повторить игру на текущем уровне. Если доступен новый уровень, то игроку также будет предоставлена возможность сыграть на нем
_Pause	namePointer, milliseconds	Вызывается для приостановки логики модуля, чтобы две карты оставались видимыми на короткое время, прежде чем перевернуть их лицевой стороной вниз или удалить, в зависимости от того, совпадают ли они. Параметр namePointer — это индекс в памяти модуля, где находится строка с именем вызываемой функции. Параметры milliseconds указывают, сколько времени пройдет перед вызовом функции

Код JavaScript использует имя элемента (например, `_GenerateCards`), чтобы указать запрошенный элемент. Однако ваш код здесь, в модуле, ссылается на импортированный элемент по индексу или по имени переменной (если вы его указываете). Вместо того чтобы работать с индексами (это может сбивать с толку), добавим имя переменной для каждого элемента импорта.

Внутри s-выражения `module` в файле `cards.wast` добавьте s-выражения `import` из листинга 11.1 для функций, указанных в табл. 11.2.

Элементы, перечисленные в табл. 11.3, будут импортированы из модуля, сгенерированного Emscripten.

Листинг 11.1. S-выражения import для элементов из кода JavaScript

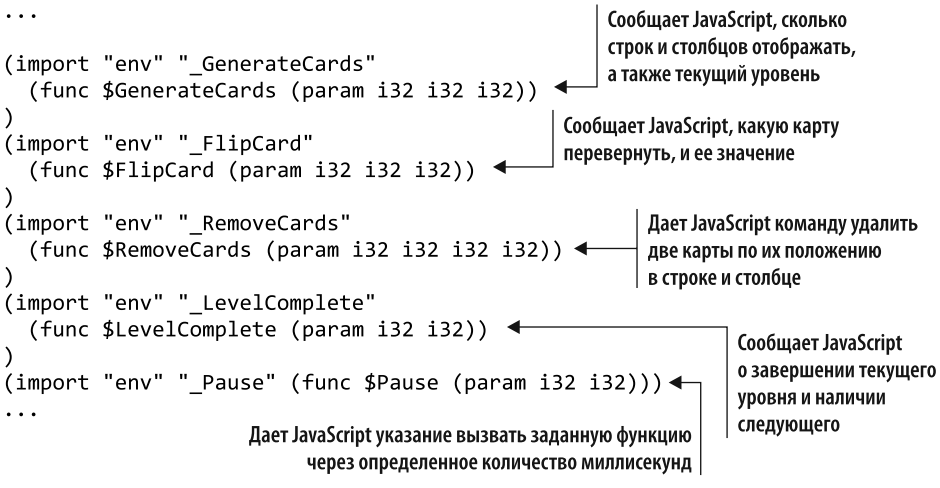


Таблица 11.3. Элементы, которые нужно импортировать из модуля, сгенерированного Emscripten

Название элемента	Тип	Параметры	Назначение
memory	Память		Линейная память модуля, сгенерированного Emscripten, которая будет выделена для совместного использования
_SeedRandomNumberGenerator	Функция		Инициализирует генератор случайных чисел
_GetRandomNumber	Функция	Диапазон	Возвращает случайное число из указанного диапазона
_malloc	Функция	Размер	Выделяет память для указанного числа байтов
_free	Функция	Указатель	Высвобождает память, выделенную для данного указателя

Импорт функций будет определен здесь так же, как и для импорта из JavaScript. Единственное, что отличает этот набор импортов, — импорт памяти.

Независимо от импортируемых элементов, первая часть узла `import` одинакова: метка `import` s-выражения, пространство имен и имя элемента. Единственное, что изменяется, — это s-выражение для импортируемого элемента.

S-выражение для памяти начинается с метки `memory`, за которой следует необязательное имя переменной, начальное количество желаемых страниц памяти

и, необязательно, максимальное количество желаемых страниц памяти. Каждая страница памяти составляет 64 Кбайт (1 Кбайт составляет 1024 байта, поэтому одна страница содержит 65 536 байт). В следующем примере будет определяться память модуля с исходной одной страницей памяти и максимум десятью страницами:

```
(memory 1 10)
```

Внутри s-выражения модуля в файле `cards.wast` добавьте s-выражения `import` из листинга 11.2 для элементов, указанных в табл. 11.3. Поместите эти узлы `import` после узла `import` для `_Pause`.

Листинг 11.2. S-выражения import для элементов из модуля, сгенерированного Emscripten

```
...

(import "env" "memory" (memory $memory 256))  <-- Память модуля
(import "env" "_SeedRandomNumberGenerator"
  (func $SeedRandomNumberGenerator)  <-- Заполняет генератор случайных чисел
)
(import "env" "_GetRandomNumber"
  (func $GetRandomNumber (param i32) (result i32))  <-- Получает случайное число из указанного диапазона
)
(import "env" "_malloc" (func $malloc (param i32) (result i32)))
(import "env" "_free" (func $free (param i32)))
...
```

После того как импорт будет задан, мы перейдем к определению некоторых глобальных переменных для логики игры.

11.1.6. Узлы global

Известный раздел «Глобальные переменные» (рис. 11.9) определяет все глобальные переменные, встроенные в модуль. Кроме того, их можно импортировать.

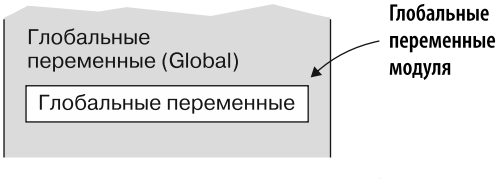


Рис. 11.9. Известный раздел «Глобальные переменные» объявляет все глобальные переменные, входящие в модуль

Глобальные переменные объявляются на уровне модуля в целях использования всеми функциями и могут быть *неизменными* (константы) или *изменяемыми*.

Они определяются с помощью узла s-выражения, который начинается с метки `global`, за которой следует необязательное имя переменной, тип переменной и затем s-выражение, содержащее значение переменной по умолчанию. Например, следующий узел `global` определяет неизменяемую (постоянную) переменную `$MAX`, которая представляет собой 32-битное целое число и имеет значение по умолчанию 25:

```
(global $MAX i32 (i32.const 25))
```

Если вам нужна изменяемая глобальная переменная, то глобальный тип заключается в s-выражение с меткой `mut`. Например, следующая глобальная переменная `$total` — изменяемое 32-разрядное число с плавающей запятой со значением по умолчанию 1,5:

```
(global $total (mut f32) (f32.const 1.5))
```

Пришло время добавить узлы `global` в игру.

Добавление узлов global в игру

Все глобальные переменные, которые потребуются в игре, будут представлены 32-битными целыми числами со значением по умолчанию, равным нулю. После s-выражений `import` и внутри s-выражения `module` добавьте следующую неизменяемую глобальную переменную в файл `cards.wast`, чтобы указать, что игра будет поддерживать не более трех уровней:

```
(global $MAX_LEVEL i32 (i32.const 3))
```

Остальные глобальные переменные будут изменяемыми, включая следующую под названием `$cards`. Она используется для хранения указателя на место в памяти модуля, в котором находится массив значений карт. Добавьте код, показанный ниже, после переменной `$MAX_LEVEL` в файле `cards.wast`:

```
(global $cards (mut i32) (i32.const 0))
```

Теперь нам понадобятся некоторые переменные для отслеживания текущего уровня игры (`$current_level`) и количества совпадений до окончания уровня (`$matches_remaining`). Кроме этого, объявим переменные `$rows` и `$columns` для хранения количества строк и столбцов, отображаемых на текущем уровне.

Добавьте код, показанный ниже, после переменной `$cards` в s-выражение `module` в файле `cards.wast`:

```
(global $current_level (mut i32) (i32.const 0))
(global $rows (mut i32) (i32.const 0))
(global $columns (mut i32) (i32.const 0))
(global $matches_remaining (mut i32) (i32.const 0))
```

Когда игрок щелкает кнопкой мыши на первой карте, необходимо запомнить положение строки и столбца карты, чтобы можно было либо перевернуть ее лицевой стороной вниз, если вторая карта не совпадает с первой, либо удалить карту в противном случае. Вдобавок необходимо отслеживать значение карты, чтобы сравнивать значение второй карты и проверять, совпадают ли они.

Когда игрок щелкает кнопкой мыши на второй карте, выполнение будет передаваться JavaScript. Это на короткое время приостанавливает игру, вследствие чего вторая карта остается видимой достаточно долго, чтобы игрок мог ее увидеть, прежде чем она будет перевернута лицом вниз или удалена. Поскольку выполняющаяся функция завершится, также необходимо запомнить позиции строки и столбца второй карты и ее значение.

В файле `cards.wast` добавьте следующий код после переменной `$matches_remaining` в s-выражение `module`:

```
(global $first_card_row (mut i32) (i32.const 0))
(global $first_card_column (mut i32) (i32.const 0))
(global $first_card_value (mut i32) (i32.const 0))
(global $second_card_row (mut i32) (i32.const 0))
(global $second_card_column (mut i32) (i32.const 0))
(global $second_card_value (mut i32) (i32.const 0))
```

Нужно предотвратить дальнейшие действия пользователя, когда выполнение модуля передается JavaScript и логика приостанавливается, прежде чем карты будут перевернуты лицевой стороной вниз или удалены. Следующая глобальная переменная послужит флагом для определения того, что в настоящее время выполнение приостановлено, пока JavaScript не обратится к модулю. В файле `cards.wast` добавьте код, показанный ниже, после переменной `$second_card_value` в s-выражение `module`:

```
(global $execution_paused (mut i32) (i32.const 0))
```

После того как глобальные переменные будут определены, перейдем к следующей области — экспорту.

11.1.7. Узлы export

Как показано на рис. 11.10, в известном разделе «Экспорт» содержится список всех элементов, которые будут возвращены в среду хоста, после того как будет создан экземпляр модуля. Это части модуля, к которым среда хоста имеет доступ. Экспорт может включать в себя элементы разделов «Функция», «Таблица», «Память» или «Глобальные переменные». Для логики нашего модуля достаточно только экспортировать функции.

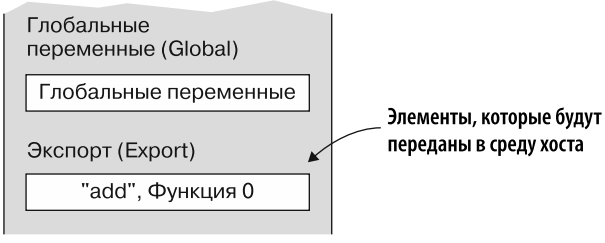


Рис. 11.10. В известном разделе «Экспорт» перечислены все элементы модуля, к которым среда хоста имеет доступ

Чтобы экспортировать элемент, нам понадобится s-выражение с меткой `export`, за которым следует имя, которое будет использоваться вызывающим объектом, а затем s-выражение, определяющее экспортируемый элемент.

Для экспорта функции s-выражение в конце узла `export` должно представлять собой `func` с индексом, отсчитываемым от нуля, или с именем переменной функции, на которую экспорт указывает в модуле. Например, следующий код будет экспортировать функцию, которую хост будет определять как `_Add`, — она указывает на функцию в модуле с именем переменной `$add`:

```
(export "_Add" (func $add))
```

Вы получили представление о том, как определяются узлы `export`, — пришло время добавить их в игру.

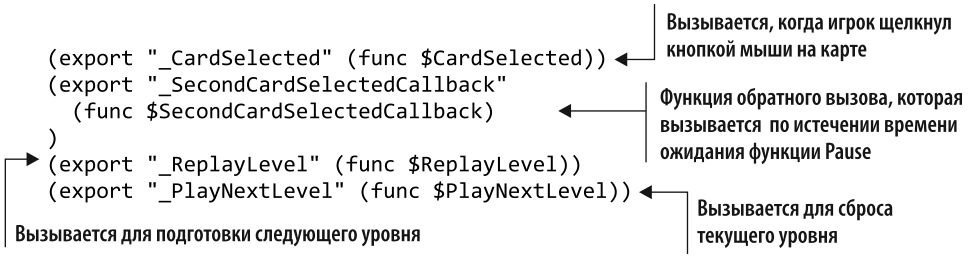
Добавление узлов export в игру

Скоро мы займемся созданием функций для логики игры. Из уже добавляемых функций необходимо экспортировать следующие:

- `$CardSelected` — вызывается кодом JavaScript всякий раз, когда игрок щелкает кнопкой мыши на карте. Логика обращается к импортированной функции `$Pause` в JavaScript, если функция была вызвана для второй карты. `$Pause` также получает указание вызвать функцию `$SecondCardSelectedCallback` после небольшой задержки;
- `$SecondCardSelectedCallback` — вызывается кодом JavaScript из функции `$Pause`. Проверяет, совпадают ли две карты или нет, и переворачивает их лицевой стороной вниз, если они не совпадают, или удаляет в случае их совпадения. Если количество оставшихся совпадений становится равным нулю, то вызывает `$LevelComplete` в JavaScript;
- `$ReplayLevel` — вызывается кодом JavaScript, когда игрок нажимает кнопку `Replay` (Повторить игру) на итоговом экране после завершения текущего уровня;

- `$PlayNextLevel` — кнопка Next Level (Следующий уровень) отображается на итоговом экране, если игрок не достиг финального уровня игры. Эта функция вызывается кодом JavaScript, когда игрок нажимает кнопку Next Level (Следующий уровень).

После s-выражений `global` внутри s-выражения `module` добавьте следующие s-выражения `export` в файл `cards.wast`:



После того как экспорт будет определен, перейдем к реализации раздела «Старт».

11.1.8. Узел start

Как показано на рис. 11.11, известный раздел «Старт» определяет функцию, которая должна быть вызвана после того, как будет создан экземпляр модуля, но перед тем, как экспортируемые элементы станут доступными для вызова. Можно указать, что функция не может быть импортирована и должна находиться в модуле.

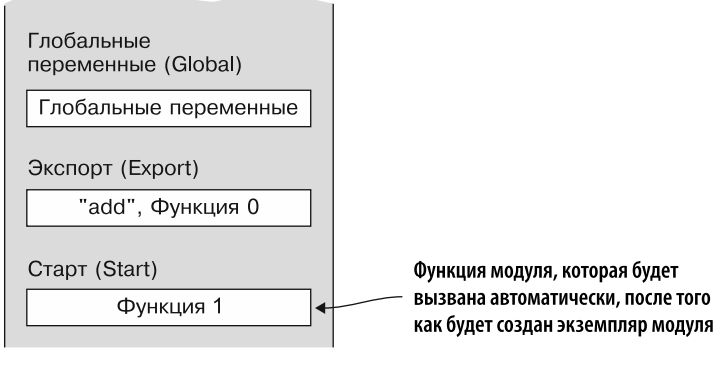


Рис. 11.11. Известный раздел «Старт» определяет функцию, которая должна быть вызвана, после того как будет создан экземпляр модуля

В данной игре функция `start` используется для инициализации глобальных переменных и памяти. Она также запускает первый уровень игры.

Раздел «Старт» определяется с помощью s-выражения с меткой `start`, за которой следует либо индекс функции, либо имя переменной. Добавьте код, показанный ниже, в файл `cards.wast` после s-выражений `export` в s-выражение `module`, чтобы функция `$main` вызывалась автоматически, после того как будет создан экземпляр модуля:

```
(start $main)
```

Следующий шаг — определение функций этого модуля и их кода.

11.1.9. Узлы code

Как показано на рис. 11.12, в двоичном формате известные разделы «Функция» (определение) и «Код» (тело) разделены. В текстовом формате определение функции и ее тело составляют одно выражение `func`. При просмотре сгенерированного Emscripten текстового формата или кода браузера функции обычно

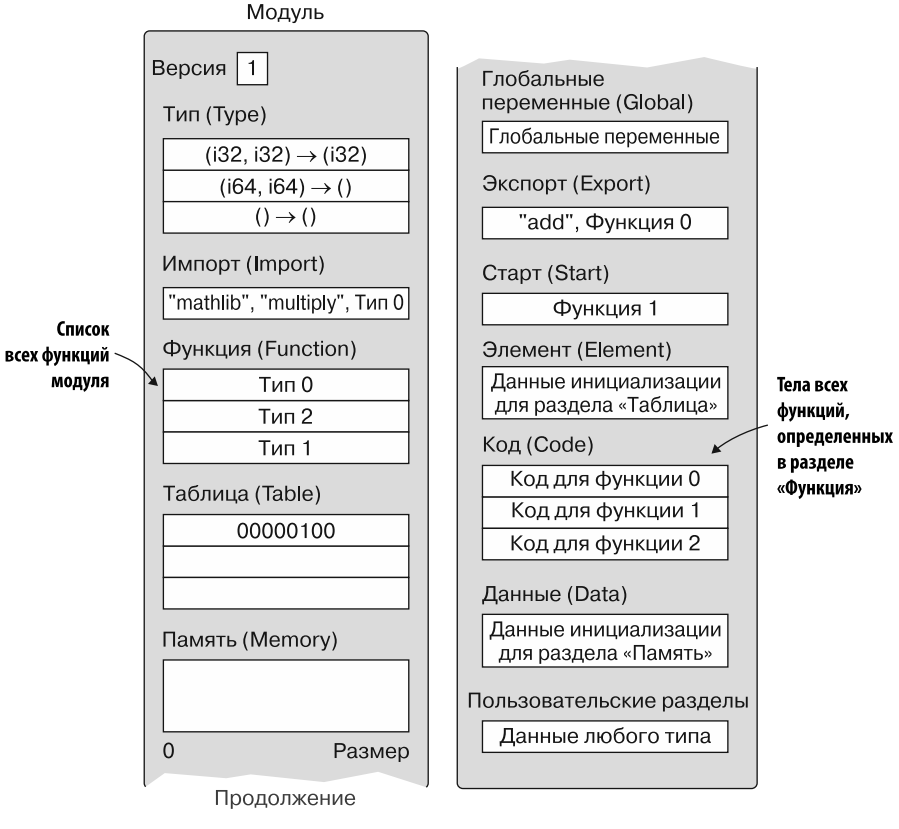


Рис. 11.12. Известные разделы «Функция» и «Код» в двоичном формате

отображаются в известном разделе «Код», поэтому в целях единообразия мы сделаем здесь так же.

Выполнение кода в WebAssembly определяется в терминах стековой машины, в которой инструкции добавляют или убирают определенное количество значений в стек и из него. Когда функция вызывается впервые, ее стек пуст. Платформа WebAssembly проверяет стек при завершении функции, чтобы гарантировать, что, например, если функция возвращает значение `i32`, то последний элемент в стеке на момент возврата значения — `i32`. Если функция ничего не возвращает, то стек должен быть пуст на момент возврата значения.

ИНФОБОКС

Текстовый формат в теле функции поддерживает стиль s-выражения, стиль стековой машины или их комбинацию. В этой главе мы будем использовать стиль стековой машины, поскольку его применяют браузеры. Примеры s-выражений см. в приложении Д, в котором приведены альтернативные способы написания операторов if и циклов.

Прежде чем вы начнете создавать функции игры, посмотрим, как взаимодействовать с переменными.

Работа с переменными

В WebAssembly существует два типа переменных: глобальные и локальные. Глобальные доступны для всех функций, тогда как локальные — только в той функции, в которой были определены.

Локальные переменные должны быть определены перед чем-либо еще в функции как s-выражение с меткой `local`, за которой следует необязательное имя переменной, а затем тип переменной. Ниже приведен пример объявления локальной переменной `f32` с именем переменной `$float`, за которым следует объявление локальной переменной `i32` без имени переменной:

```
(local $float f32)
(local i32)
```

Если не указывать имя переменной, то на нее можно будет ссылаться, используя индекс, отсчитываемый от нуля. При работе с локальными переменными следует помнить, что параметры функции также считаются локальными и стоят первыми в порядке индекса.

Чтобы присвоить переменной значение, его нужно предварительно добавить в стек. Затем применить инструкцию `set_local` или `tee_local`, чтобы извлечь значение из стека и установить значение локальной переменной. Разница между этими инструкциями заключается в том, что `tee_local` также возвращает установленное значение. Для глобальной переменной инструкция `set_global` используется так же, как и `set_local`.

Например, следующий фрагмент кода помещает значение `10.3` в стек, а затем вызывает инструкцию `set_local` для переменной `$float`. Инструкция извлечет верхнее значение из стека и поместит его в указанную переменную:

```
f32.const 10.3
set_local $float
```

Чтобы получить значение переменной и поместить его в стек, используется инструкция `get_local` для локальных переменных и `get_global` для глобальных. Например, если функция имеет параметр `$param0`, то следующий код поместит его значение в стек:

```
get_local $param0
```

СПРАВКА

В этой главе используются инструкции set_local, tee_local, get_local, set_global и get_global, поскольку браузеры все еще задействуют данный формат. Однако спецификация WebAssembly была скорректирована, чтобы в ней применялись local.set, local.tee, local.get, global.set и global.get. Новый формат вызывается точно так же, как и старый. Сейчас Emscripten выдает файл .wast в новом формате, а WebAssembly Binary Toolkit теперь может принимать код текстового формата и использовать любой формат. Новые инструкции для переменных можно найти по адресу http://mng.bz/xljX.

Разобравшись, как работают переменные, перейдем к созданию первого функционального узла для логики игры — функции `$InitializeRowsAndColumns`.

Функция $InitializeRowsAndColumns

Функция `$InitializeRowsAndColumns` имеет единственный параметр `i32` с именем `$level` и не возвращает значения. Цель ее вызова — задать соответствующие значения глобальных переменных `$rows` и `$columns` на основе полученного параметра уровня.

Каждый уровень имеет разную комбинацию строк и столбцов для карт, поэтому функция должна определить, какой именно уровень был запрошен. Чтобы проверить, равно ли значение параметра `1` (единице), поместим значение параметра в стек, а затем добавим в стек `i32.const 1`. Чтобы определить, равны ли два значения в стеке, вызовем инструкцию `i32.eq`, которая удаляет два верхних элемента из стека, проверяет, равны ли они, а затем помещает результат в стек (`1` для истины, `0` для лжи), как показано ниже:

```
get_local $level
i32.const 1
i32.eq
```

В стек будет помещено значение 1, если значение переменной $level равно 1. В противном случае в стек будет помещено значение 0

Поместив в стек логическое значение, будем использовать оператор `if`, чтобы проверить, истинно ли логическое значение, и если да, то установить для каждого значения `$rows` и `$column` значение `i32.const 2`. Оператор `if` удалит верхний элемент из стека для вычисления. Этот оператор считает нулевое значение ложным, а любое ненулевое значение — истинным. Следующий фрагмент кода расширяет логику предыдущего фрагмента, добавляя оператор `if`:

```
get_local $level
i32.const 1
i32.eq
if
end
```

Если верхнее значение в стеке не равно 0, то будет запущен код в данном блоке

Код, показанный в предыдущем фрагменте, будет повторяться три раза, по одному разу для каждого проверяемого уровня. Значение `i32.const` будет изменено на `2` при проверке того, равен ли указанный уровень двум, и на `3` при проверке того, равен ли указанный уровень трем.

Установите следующие глобальные значения `$rows` и `$columns` в зависимости от указанного уровня:

- уровень 1: оба равны `i32.const 2`;
- уровень 2: `$rows` — `i32.const 2`, `$columns` — `i32.const 3`;
- уровень 3: `$rows` — `i32.const 2`, `$columns` — `i32.const 4`.

В игре предусмотрено шесть уровней, но для упрощения кода в данной функции определены только первые три. Добавьте код, показанный в листинге 11.3, после узла `start` в файле `cards.wast`.

Листинг 11.3. Функция $InitializeRowsAndColumns для файла cards.wast

```
...

(func $InitializeRowsAndColumns (param $level i32)
  get_local $level
  i32.const 1
  i32.eq

  if
    i32.const 2
    set_global $rows
    i32.const 2
    set_global $columns
  end

  get_local $level
  i32.const 2
```

Добавляет значение параметра в стек

Добавляет 1 в стек

Удаляет два верхних значения из стека, проверяет, равны ли они, и помещает результат в стек

Удаляет верхний элемент из стека; если значение истинно, задает значение глобальных переменных

Добавляет 2 в стек

Удаляет верхний элемент из стека и помещает его в глобальную переменную $rows

Удаляет верхний элемент из стека и помещает его в глобальную переменную $columns

```
  i32.eq
  if
    i32.const 2
    set_global $rows

    i32.const 3
    set_global $columns
  end

  get_local $level
  i32.const 3
  i32.eq
  if
    i32.const 3
    set_global $rows

    i32.const 4
    set_global $columns
end
)
```

Если был запрошен уровень 2, то задает значение глобальной переменной $rows равным 2

Если был запрошен уровень 2, то задает значение глобальной переменной $columns равным 3

Если был запрошен уровень 3, то задает значение глобальной переменной $rows равным 3

Если был запрошен уровень 3, то задает значение глобальной переменной $columns равным 4

Следующий узел `func`, который нужно будет определить, — функция `$Reset-SelectedCardValues`.

Функция $ResetSelectedCardValues

Функция `$ResetSelectedCardValues` не имеет параметров и не возвращает значение. Она вызывается для того, чтобы задать глобальные переменные для первой и второй карт, на которых щелкнули кнопкой мыши, равными `-1`. Установка значений этих карт равными `-1` дает понять остальной логике игры, что все карты в настоящее время закрыты. Добавьте код, показанный в листинге 11.4, после узла `$InitializeRowsAndColumns` в файле `cards.wast`.

Листинг 11.4. Функция $ResetSelectedCardValues для файла cards.wast

```
...
(func $ResetSelectedCardValues
  i32.const -1
  set_global $first_card_row

  i32.const -1
  set_global $first_card_column

  i32.const -1
  set_global $first_card_value

  i32.const -1
  set_global $second_card_row

  i32.const -1
  set_global $second_card_column
```

```
  i32.const -1
  set_global $second_card_value
)
```

Следующий узел `func`, который нужно определить, — функция `$InitializeCards`.

Функция $InitializeCards

Функция `$InitializeCards` имеет параметр `i32` с именем `$level` и не возвращает значения. Эта функция вызывается в целях установки соответствующих значений глобальных переменных в соответствии с полученным параметром `$level`, создания и заполнения массива `$cards` и последующего перемешивания массива.

Локальные переменные должны быть определены в функции перед любым другим кодом, поэтому первое, что необходимо добавить в функцию, — локальная переменная `i32` с именем `$count`, которая будет заполнена позже. В следующем фрагменте кода показано определение локальной переменной:

```
(local $count i32)
```

Следующее, что делает функция, — помещает полученный параметр `$level` в стек, а затем вызывает `set_global`, чтобы извлечь значение из стека и поместить его в глобальную переменную `$current_level`:

```
get_local $level
set_global $current_level
```

Затем значение параметра `$level` снова помещается в стек и вызывается функция `$InitializeRowsAndColumns`, чтобы глобальные переменные `$rows` и `$columns` были установлены соответствующим образом в зависимости от запрошенного уровня. Поскольку функция имеет единственный параметр, WebAssembly удалит верхнее значение из стека (значение `level`) и передаст его функции, как показано ниже:

```
get_local $level
call $InitializeRowsAndColumns
```

Чтобы глобальные переменные первой и второй карты были сброшены до значения `-1`, код вызывает функцию `$ResetSelectedCardValues`. Она не имеет параметров, поэтому в стек для вызова функции ничего помещать не нужно:

```
call $ResetSelectedCardValues
```

Затем функция определяет, сколько карт необходимо для текущего уровня, на основе значений глобальных переменных `$rows` и `$columns`. Эти значения глобальных переменных помещаются в стек, после чего вызывается инструкция `i32.mul`. Она извлекает два верхних элемента из стека, умножает их значения и помещает результат обратно в стек. Как только результат появится в стеке, вызывается `set_local`, чтобы поместить значение в переменную `$count`. Вызов `set_local`

удаляет верхний элемент из стека и помещает его в указанную переменную. Ниже показан код, определяющий количество карт на текущем уровне:

```
get_global $rows
get_global $columns
i32.mul
set_local $count
```

Следующий шаг — определение значения `$matches_remaining` путем деления `$count` на 2. Значения `$count` и `i32.const 2` помещаются в стек, а затем вызывается инструкция `i32.div_s`. Она извлекает из стека два верхних элемента, делит их друг на друга и помещает результат обратно в стек. Затем вызывается инструкция `set_global`, чтобы удалить верхний элемент из стека и поместить его значение в глобальную переменную `$matches_remaining`:

```
get_local $count
i32.const 2
i32.div_s
set_global $matches_remaining
```

Следующее, что должно произойти в функции, — выделение блока памяти для хранения количества значений `i32` на основе значения в `$count`. Поскольку каждое значение `i32` составляет 4 байта, значение `$count` необходимо умножить на 4, чтобы получить общее количество выделяемых байтов. Можно использовать инструкцию `i32.mul`, но более эффективно применить инструкцию `i32.shl` (сдвиг влево). Сдвиг влево на 2 аналогичен умножению на 4.

После того как будет определено общее количество байтов, вызывается функция `$malloc`, импортированная из модуля, созданного Emscripten, для выделения этого количества. Функция `$malloc` вернет индекс памяти, с которого начинается выделенный блок памяти. Затем вы вызовете инструкцию `set_global`, чтобы поместить это значение в переменную `$cards`.

В следующем фрагменте кода показано определение количества байтов по значению `$count`, передача его в функцию `$malloc` и помещение результата в переменную `$cards`:

```
get_local $count
i32.const 2
i32.shl
call $malloc
set_global $cards
```

Выделив блок памяти для массива `$cards`, вызовем функцию `$PopulateArray`, передав ей количество карт на текущем уровне, как показано ниже. Функция добавит пары значений в массив `$cards` в зависимости от количества карт для текущего уровня (например, 0, 0, 1, 1, 2, 2):

```
get_local $count
call $PopulateArray
```

Наконец, функция вызовет `$ShuffleArray`, чтобы перемешать содержимое массива `$cards`:

```
get_local $count
call $ShuffleArray
```

Собрав все это воедино, добавьте код, показанный в листинге 11.5, после узла `$ResetSelectedCardValues` в файле `cards.wast`.

Листинг 11.5. Функция $InitializeCards для файла cards.wast

```
...

(func $InitializeCards (param $level i32)
  (local $count i32)

  get_local $level
  set_global $current_level

  get_local $level
  call $InitializeRowsAndColumns

  call $ResetSelectedCardValues

  get_global $rows
  get_global $columns
  i32.mul
  set_local $count

  get_local $count
  i32.const 2
  i32.div_s
  set_global $matches_remaining

  get_local $count
  i32.const 2
  i32.shl
  call $malloc
  set_global $cards

  get_local $count
  call $PopulateArray

  get_local $count
  call $ShuffleArray
)
```

Запоминает запрошенный уровень

Устанавливает значения глобальных переменных строк и столбцов на основе текущего уровня

Проверяет, что значения первой и второй карты сброшены

Определяет, сколько карт нужно на текущем уровне

Определяет, сколько пар карт нужно на текущем уровне

Выполняет сдвиг на 2 влево, поскольку все элементы массива представлены 32-битными целыми числами (по 4 байта каждый)

Выделяет необходимое количество памяти, вызывая функцию malloc

Заполняет массив парами значений

Перемешивает массив

Следующий узел `func`, который нужно определить, — функция `$PopulateArray`.

Функция $PopulateArray

Пройдитесь по всем элементам массива, как показано в листинге 11.6, добавляя пары значений в зависимости от количества карт на текущем уровне (например, 0, 0, 1, 1, 2, 2).

Листинг 11.6. Функция $PopulateArray для файла cards.wast

```
...

(func $PopulateArray (param $array_length i32)
  (local $index i32)
  (local $card_value i32)

  i32.const 0
  set_local $index

  i32.const 0
  set_local $card_value

  loop $while-populate
    get_local $index
    call $GetMemoryLocationFromIndex
    get_local $card_value
    i32.store
                                  Устанавливает значение памяти
                                  по $index в значение $card_value
    get_local $index
    i32.const 1
    i32.add
    set_local $index
                                  Увеличивает индекс на 1

    get_local $index
    call $GetMemoryLocationFromIndex
    get_local $card_value
    i32.store
                                  Устанавливает значение памяти
                                  по $index в значение $card_value
    get_local $card_value
    i32.const 1
    i32.add
    set_local $card_value
                                  Увеличивает $card_value
                                  на 1 для следующего цикла
    get_local $index
    i32.const 1
    i32.add
    set_local $index
                                  Увеличивает индекс на 1
                                  для следующего цикла
    get_local $index
    get_local $array_length
    i32.lt_s
    if
      br $while-populate
                                  Если индекс меньше, чем $array_length,
                                  цикл повторяется
    end
  end $while-populate
)
```

Следующий узел `func`, который нужно определить, — функция `$GetMemoryLocationFromIndex`.

Функция $GetMemoryLocationFromIndex

Функция `$GetMemoryLocationFromIndex` имеет параметр `i32` с именем `$index` и возвращает значение `i32`. Ее вызов позволяет определить местоположение индекса в массиве `$cards` в памяти.

Функция помещает в стек значение параметра (`$index`), а также значение `i32.const 2`. Затем вызывает инструкцию `i32.shl` (сдвиг влево), которая удаляет два верхних значения из стека, сдвигает значение `$index` на 2 (получая тот же результат, что и при умножении его на 4) и помещает результат обратно в стек.

Затем функция вызывает `get_global` для `$cards`, чтобы поместить начальную позицию массива `$cards` в памяти в стек. Затем вызывается инструкция `i32.add`; она извлекает из стека два верхних элемента, складывает их и помещает результат обратно в стек. Функция будет возвращать значение, поэтому результат операции `i32.add` остается в стеке и передается вызывающей стороне.

Добавьте код, показанный ниже, после узла `$PopulateArray` в файле `cards.wast`:

```
(func $GetMemoryLocationFromIndex (param $index i32) (result i32)
  get_local $index
  i32.const 2
  i32.shl   <-------- Сдвигает значение индекса на 2 влево

  get_global $cards
  i32.add   <--------+ Добавляет начальную позицию
)                    | массива к позиции по индексу
```

Следующий узел `func`, который нужно определить, — функция `$ShuffleArray`.

Функция $ShuffleArray

Функция `$ShuffleArray` имеет параметр `i32` с именем `$array_length` и не возвращает значения. Она вызывается для перемешивания содержимого массива `$cards`.

СПРАВКА

Тип перемешивания, который будет использоваться для этого массива, — тасование Фишера — Йетса. Дополнительную информацию можно найти по адресу https://gist.github.com/sundeepblue/10501662.

Эта функция сначала определяет несколько локальных переменных для использования в следующем цикле. Затем вызывает функцию `$SeedRandomNumberGenerator`,

которая была импортирована из модуля, созданного Emscripten, для заполнения генератора случайных чисел.

Значение `$index` инициализируется на 1 меньше, чем значение `$array_length`, поскольку цикл по картам будет идти от конца массива к началу. Затем запускается цикл — он будет продолжаться, пока значение `$index` не станет равным нулю.

Внутри цикла вызывается функция `$GetRandomNumber`, которая была импортирована из модуля, созданного Emscripten, для получения случайного числа из указанного диапазона. Указанный диапазон определяется от текущего индекса, равного единице, — на его основе будет получено число от `1` до `$index + 1`. Полученное случайное число затем помещается в локальную переменную `$card_to_swap`:

```
get_local $index
i32.const 1
i32.add
call $GetRandomNumber
set_local $card_to_swap
```

Добавляет 1 к значению в $index, чтобы получить индекс, начинающийся с единицы

После того как будет определен индекс случайной карты для замены, определяется текущий индекс места карты в памяти и индекс карты для замены — эти значения помещаются в локальные переменные `$memory_location1` и `$memory_location2` соответственно.

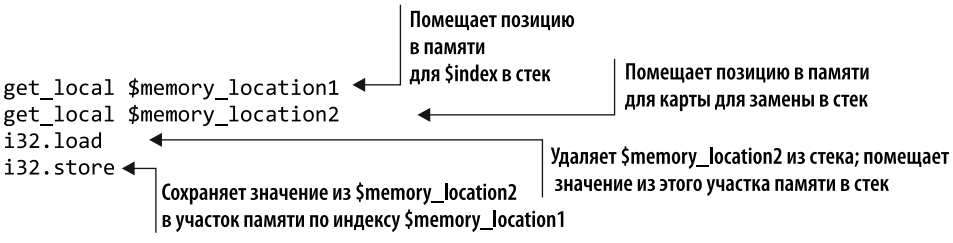
После того как две ячейки памяти будут найдены, значение текущего индекса (`$memory_location1`) считывается из памяти путем вызова `i32.load`. Эта инструкция удаляет верхний элемент — ячейку памяти — из стека и считывает значение `i32` из этой ячейки памяти, помещая его в стек. Затем функция поместит значение в локальную переменную `$card_value`, чтобы оно не было потеряно, пока данные из `$memory_location2` помещаются в `$memory_location1`, как показано ниже:

```
get_local $memory_location1
i32.load
set_local $card_value
```

Следующий фрагмент может сбить с толку. Код помещает значение из `$memory_location1` в стек (текущий индекс), а затем помещает значение из `$memory_location2` (индекс карты для замены) в стек. Затем он вызывает `i32.load`, который извлекает верхний элемент из стека (`$memory_location2` — индекс карты для замены), считывает значение из этой ячейки памяти и помещает ее значение в стек.

Поскольку `$memory_location1` (текущий индекс) уже находится в стеке, а теперь там же находится и значение из `$memory_location2`, код может вызвать инструкцию `i32.store`. Данный вызов удалит два верхних элемента из стека и поместит

значение в память. Самый верхний элемент — значение, которое нужно сохранить, а следующий элемент — место в памяти для хранения значения:



Теперь, когда значение из `$memory_location2` находится в `$memory_location1`, код помещает значение, которое было в `$memory_location1`, в `$memory_location2`, как показано ниже:

```
get_local $memory_location2
get_local $card_value
i32.store
```

Затем цикл уменьшает значение `$index` на 1. Если значение `$index` все еще больше нуля, то цикл повторяется.

Собрав все это вместе, добавьте код, показанный в листинге 11.7, после узла `$PopulateArray` в файле `cards.wast`.

Листинг 11.7. Функция $ShuffleArray для файла cards.wast

```
...

(func $ShuffleArray (param $array_length i32)
  (local $index i32)
  (local $memory_location1 i32)
  (local $memory_location2 i32)
  (local $card_to_swap i32)
  (local $card_value i32)

  call $SeedRandomNumberGenerator
  get_local $array_length
  i32.const 1
  i32.sub
  set_local $index

  loop $while-shuffle
    get_local $index
    i32.const 1
    i32.add
    call $GetRandomNumber
    set_local $card_to_swap
```

```
    get_local $index
    call $GetMemoryLocationFromIndex
    set_local $memory_location1

    get_local $card_to_swap
    call $GetMemoryLocationFromIndex
    set_local $memory_location2

    get_local $memory_location1
    i32.load
    set_local $card_value

    get_local $memory_location1
    get_local $memory_location2
    i32.load
    i32.store

    get_local $memory_location2
    get_local $card_value
    i32.store

    get_local $index
    i32.const 1
    i32.sub
    set_local $index

    get_local $index
    i32.const 0
    i32.gt_s
    if
      br $while-shuffle
    end
  end $while-shuffle
)
```

Определяет место в памяти на основе индекса

Определяет место в памяти на основе индекса card_to_swap

Получает значение карты из памяти по текущему индексу в массиве

Удаляет $memory_location2 и помещает значение из этого участка памяти в стек

Помещает значение из $memory_location2 в $memory_location1

Помещает значение карты в память туда, где было значение card_to_swap

Уменьшает индекс на 1 для следующего цикла

Если индекс все еще больше нуля, то цикл повторяется

Следующий узел `func`, который нужно определить, — функция `$PlayLevel`.

Функция $PlayLevel

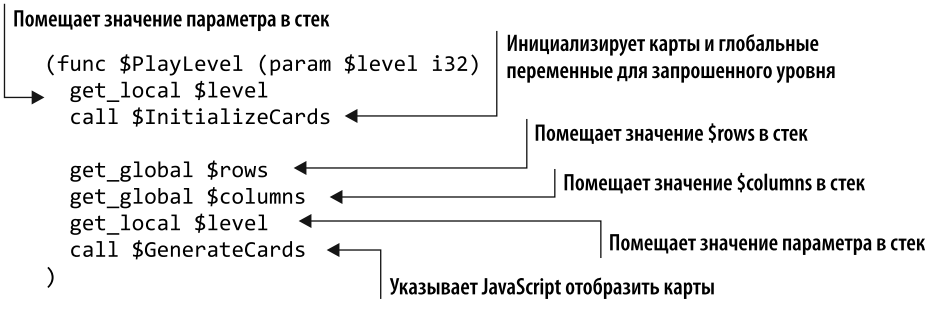
Функция `$PlayLevel` имеет параметр `i32` с именем `$level` и не возвращает значение. Эта функция вызывается для того, чтобы инициализировать карты, а затем показать их игроку.

Чтобы инициализировать карты, поместите значение параметра `$level` в стек, а затем вызовите функцию `InitializeCards`. Поскольку функция ожидает один параметр, то верхний элемент в стеке извлекается и передается в качестве параметра функции.

Затем нужно вызвать функцию `$GenerateCards` в JavaScript, чтобы игроку показывалось правильное количество карт на текущем уровне. Для этого в стек

помещаются глобальные значения `$rows` и `$columns`, а затем значение параметра `$level`. Далее вызывается функция `$GenerateCards`. Она ожидает три параметра, поэтому три верхних элемента будут извлечены из стека и переданы в параметры функции.

Добавьте код из следующего фрагмента после функции `$ShuffleArray` в файле `cards.wast`:



Следующий узел `func`, который нужно будет определить, — функция `$GetCardValue`.

Функция $GetCardValue

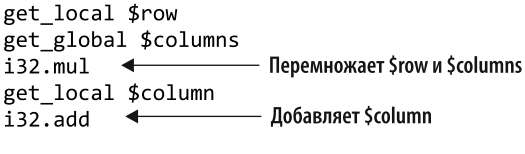
Функция `$GetCardValue` принимает два параметра `i32` (`$row` и `$column`) и возвращает результат `i32`. Она вызывается в целях получения значения карты, связанного с картой на определенной позиции строки и столбца.

Следующее выражение используется для определения индекса в массиве `$cards`, в котором находится запрошенное значение строки и столбца:

```
row * columns + column
```

В следующем фрагменте показан код в текстовом формате, реализующий эту формулу. Значение параметра `$row` помещается в стек, а затем в него же помещается глобальная переменная `$columns`. Инструкция `i32.mul` извлекает два верхних элемента из стека, перемножает их, а затем помещает результат в стек.

Значение параметра `$column` помещается в стек, а затем вызывается инструкция `i32.add`, которая извлекает два верхних элемента из стека, складывает их и помещает результат в стек, предоставляя индекс в массиве для получения значения карты:



После того как индекс массива будет определен, необходимо сдвинуть индекс влево на 2 (умножить на 4), поскольку каждый индекс представляет собой 4-байтовое 32-битное целое число. Затем начальная позиция массива `$cards` в памяти добавляется к скорректированному индексу, чтобы получить место в памяти модуля, в котором находится этот индекс. Теперь, когда индекс памяти находится в стеке, вызывается инструкция `i32.load`, которая извлекает верхний элемент из стека, считывает элемент из данной ячейки памяти и помещает значение в стек. Поскольку эта функция возвращает результат `i32`, то вы просто оставляете результат вызова `i32.load` в стеке, и он будет возвращен вызывающей функции, когда она завершится.

Добавьте код, показанный в листинге 11.8, после функции `$PlayLevel` в файле `cards.wast`.

Листинг 11.8. Функция $GetCardValue для файла cards.wast

```
...

(func $GetCardValue (param $row i32) (param $column i32) (result i32)
  get_local $row
  get_global $columns
  i32.mul
  get_local $column
  i32.add

  i32.const 2
  i32.shl
  get_global $cards
  i32.add
  i32.load
)
```

Перемножает значения $row и $columns

Добавляет значение $column к результату умножения

Сдвигает значение индекса на 2 влево (умножает на 4), поскольку каждый элемент — это 32-битное целое число

Читает значение из памяти; оставляет его в стеке, чтобы вернуть вызывающей функции

Добавляет начальную позицию массива $cards к позиции по индексу

Следующий узел `func`, который нужно определить, — функция `$CardSelected`.

Функция $CardSelected

Функция `$CardSelected` принимает два параметра `i32` (`$row` и `$column`) и не возвращает значение. Ее вызывает код JavaScript, когда игрок щелкает кнопкой мыши на карте.

Как показано в следующем фрагменте кода, перед началом вычислений эта функция проверяет, не приостановлено ли выполнение. Оно будет приостановлено, если игрок только что щелкнул на второй карте, — модуль определяет небольшую задержку, прежде чем перевернуть карты лицом вниз или удалить их. Если выполнение приостановлено, то функция завершается вызовом оператора `return`:

```
get_global $execution_paused
i32.const 1
i32.eq
if
  return
end
```

Если выполнение не приостановлено, то функция определит значение карты для переданных через параметры переменных `$row` и `$column`, вызвав функцию `$GetCardValue`. Определенное значение карты помещается в локальную переменную `$card_value`, как показано ниже:

```
get_local $row
get_local $column
call $GetCardValue
set_local $card_value
```

Затем функция вызовет `$FlipCard` в JavaScript, чтобы карта, на которой был сделан щелчок, перевернулась лицевой стороной вверх:

```
get_local $row
get_local $column
get_local $card_value
call $FlipCard
```

Затем код проверяет, равно ли `$first_card_row` значению `-1`. Если да, то первая карта еще не открыта и выполняется блок `then` оператора `if`. Если нет, то это означает, что первая карта уже открыта, поэтому выполняется блок `else` оператора `if`, как показано ниже:

```
get_global $first_card_row
i32.const -1
i32.eq
if
```

Значение $first_card_row равно –1. Первая карта еще не открыта

```
else
```

Значение $first_card_row не равно –1. Первая карта открыта

```
end
```

В блоке `then` оператора `if` значения `$row`, `$column` и `$card_value` помещаются в глобальные переменные `$first_card_row`, `$first_card_column` и `$first_card_value` соответственно.

В блоке `else` оператора `if` код сначала проверяет, принадлежат ли значения `$row` и `$column` к первой карте, вызывая функцию `$IsFirstCard`. Если игрок снова щелкнул на той же карте, то функция завершится, как показано ниже:

```
get_local $row
get_local $column
call $IsFirstCard
if
  return
end
```

Если игрок щелкнул кнопкой мыши на другой карте, то ветвь `else` помещает значения `$row`, `$column` и `$card_value` в глобальные переменные `$second_card_row`, `$second_card_column` и `$second_card_value` соответственно. Затем код ветвления `else` присваивает переменной `$execution_paused` значение `i32.const 1`, чтобы показать, что выполнение приостановлено и функция не должна реагировать на щелчки кнопкой мыши, пока выполнение не будет возобновлено.

Наконец, как показано в следующем фрагменте, код в ветви `else` помещает в стек сначала значение `i32.const 1024`, а затем `i32.const 600`. Значение `1024` — это ячейка памяти для строки `"SecondCardSelectedCallback"`, которую вы укажете при определении известного раздела «Данные» далее в этой главе. Значение `600` — это количество миллисекунд, на которое код JavaScript приостановит выполнение.

После того как два значения будут помещены в стек, вызывается функция `$Pause` в JavaScript. Она ожидает два параметра, поэтому два верхних элемента в стеке извлекаются и передаются в качестве параметров функции:

```
i32.const 1024
i32.const 600
call $Pause
```

Собрав все вместе, добавьте код, показанный в листинге 11.9, после функции `$GetCardValue` в файле `cards.wast`.

Листинг 11.9. Функция $CardSelected для файла cards.wast

```
...

(func $CardSelected (param $row i32) (param $column i32)
  (local $card_value i32)

  get_global $execution_paused
  i32.const 1
  i32.eq
  if
    return
  end

  get_local $row
  get_local $column
  call $GetCardValue
  set_local $card_value
```

Игнорирует щелчки кнопкой мыши, если игра приостановлена

Получает значение карты для указанных строки и столбца

```
  get_local $row
  get_local $column
  get_local $card_value
  call $FlipCard

  get_global $first_card_row
  i32.const -1
  i32.eq
  if
    get_local $row
    set_global $first_card_row

    get_local $column
    set_global $first_card_column

    get_local $card_value
    set_global $first_card_value
  else
    get_local $row
    get_local $column
    call $IsFirstCard
    if
      return
    end

    get_local $row
    set_global $second_card_row

    get_local $column
    set_global $second_card_column

    get_local $card_value
    set_global $second_card_value

    i32.const 1
    set_global $execution_paused

    i32.const 1024
    i32.const 600
    call $Pause
  end
)
```

Дает JavaScript указание перевернуть данную карту

Если игрок еще не щелкнул ни на одной карте…

…запоминает данные карты, на которой был сделан щелчок

Первая карта уже показана

Если игрок снова щелкнул кнопкой мыши на первой карте, то выйти из функции

Запоминает данные второй карты

Не отвечает на щелчки кнопкой мыши, пока не выполнится функция обратного вызова Pause в модуле

Место строки "SecondCardSelectedCallback" в памяти

Промежуток времени перед вызовом функции $SecondCardSelectedCallback в JavaScript

Вызывает функцию $Pause в JavaScript

Следующий узел `func`, который нужно будет определить, — функция `$IsFirstCard`.

Функция $IsFirstCard

Функция `$IsFirstCard` принимает два параметра `i32` (`$row` и `$column`) и возвращает результат `i32`. Цель ее вызова — определить, относятся ли значения `$row` и `$column` к первой карте, показываемой пользователю.

Функция сначала проверяет, соответствует ли значение параметра `$row` глобальному значению `$first_card_row`, и помещает результат в локальную переменную `$rows_equal`. Таким же образом функция проверяет соответствие значения параметра `$column` глобальному значению `$first_card_column` и помещает результат в локальную переменную `$columns_equal`.

Затем функция помещает значения `$rows_equal` и `$columns_equal` в стек и вызывает инструкцию `i32.and`. Эта инструкция извлекает из стека два верхних элемента и выполняет поразрядную операцию И над значениями, чтобы определить, равны ли они друг другу; затем результат помещается обратно в стек. Поскольку функция возвращает результат `i32`, то вы оставляете результат `i32.and` на стеке — он будет возвращен вызывающей функции, когда вызванная функция завершится.

Добавьте код, показанный в листинге 11.10, после функции `$CardSelected` в файле `cards.wast`.

Листинг 11.10. Функция $IsFirstCard для файла cards.wast

```
...

(func $IsFirstCard (param $row i32) (param $column i32) (result i32)
  (local $rows_equal i32)
  (local $columns_equal i32)

  get_global $first_card_row
  get_local $row
  i32.eq
  set_local $rows_equal

  get_global $first_card_column
  get_local $column
  i32.eq
  set_local $columns_equal

  get_local $rows_equal
  get_local $columns_equal
  i32.and
)
```

Определяет, равна ли строка первой карты текущей строке

Определяет, равен ли столбец первой карты текущему столбцу

Побитовое И для определения равенства строк и столбцов соответственно

Следующий узел `func`, который нужно будет определить, — функция `$SecondCardSelectedCallback`.

Функция $SecondCardSelectedCallback

Функция `$SecondCardSelectedCallback` не имеет параметров и не возвращает значение. Она вызывается функцией `$Pause` в JavaScript по истечении времени ожидания. Функция `$SecondCardSelectedCallback` проверяет, совпадают ли две

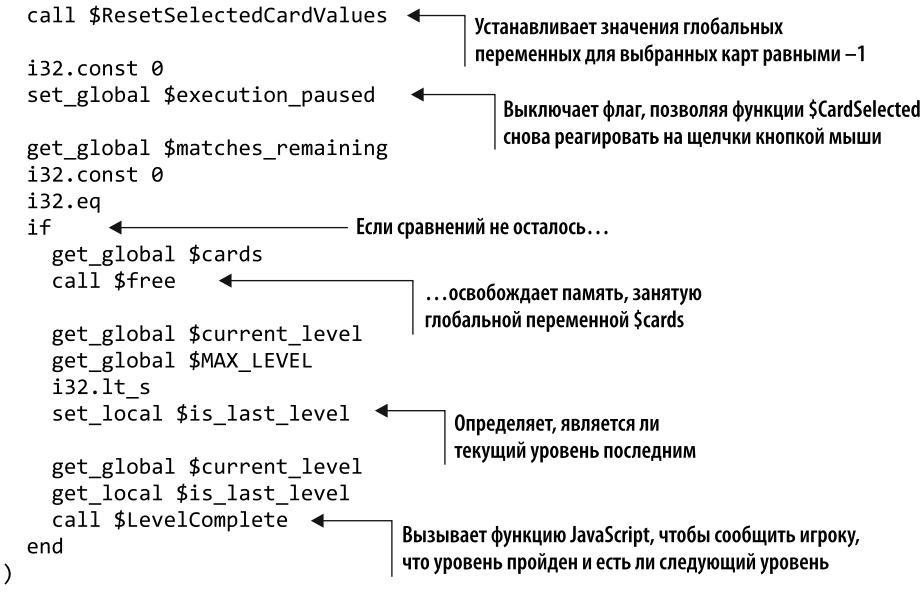
выбранные карты. Если да, то вызывается функция `$RemoveCards` в JavaScript, чтобы скрыть обе карты, а затем глобальная переменная `$match_remaining` уменьшается на 1. Если карты не совпадают, то для каждой из них вызывается функция `$FlipCard` в JavaScript, которая переворачивает их лицевой стороной вниз. Затем глобальные переменные, указывающие, какие карты были открыты, сбрасываются, а для переменной `$execution_paused` устанавливается значение `0` (ноль), что указывает на то, что выполнение кода модуля больше не приостановлено.

Затем функция проверяет, равно ли `$match_remaining` значению `0` (нулю), что указывает на завершение уровня. Если да, то память для массива `$cards` освобождается путем вызова функции `$free`, импортированной из модуля, созданного Emscripten. Затем вызывается функция `$LevelComplete` в JavaScript, чтобы сообщить игроку, что он прошел уровень.

Добавьте код, показанный в листинге 11.11, после функции `$IsFirstCard` в файле `cards.wast`.

Листинг 11.11. Функция $SecondCardSelectedCallback для файла cards.wast

```
...

(func $SecondCardSelectedCallback
  (local $is_last_level i32)

  get_global $first_card_value
  get_global $second_card_value
  i32.eq
  if                                    ← Если две выбранные карты совпадают…
    get_global $first_card_row
    get_global $first_card_column
    get_global $second_card_row
    get_global $second_card_column
    call $RemoveCards                   ← …вызывается JavaScript-код, чтобы скрыть обе карты

    get_global $matches_remaining
    i32.const 1
    i32.sub
    set_global $matches_remaining       ← Уменьшает значение глобальной переменной на 1
  else                                  ← Значения двух карт не совпадают
    get_global $first_card_row
    get_global $first_card_column       ┐ Дает JavaScript указание перевернуть
    i32.const -1                        │ первую карту лицевой стороной вниз
    call $FlipCard                      ┘

    get_global $second_card_row
    get_global $second_card_column      ┐ Дает JavaScript указание перевернуть
    i32.const -1                        │ вторую карту лицевой стороной вниз
    call $FlipCard                      ┘
  end
```



Следующий узел `func`, который нужно будет определить, — функция `$ReplayLevel`.

Функция $ReplayLevel

Функция `$ReplayLevel` не имеет параметров, не возвращает значение и вызывается JavaScript, когда игрок нажимает кнопку Replay (Повторить игру). Эта функция просто передает глобальную переменную `$current_level` функции `$PlayLevel`.

Добавьте код, показанный ниже, после функции `$SecondCardSelectedCallback` в файле `cards.wast`:

```
(func $ReplayLevel
  get_global $current_level
  call $PlayLevel
)
```

Следующий узел `func`, который нужно определить, — функция `$PlayNextLevel`.

Функция $PlayNextLevel

Функция `$PlayNextLevel` не имеет параметров, не возвращает значение и вызывается из JavaScript, когда игрок нажимает кнопку Next Level (Следующий уровень). Эта функция вызывает функцию `$PlayLevel`, передавая ей значение, которое на 1 больше значения глобальной переменной `$current_level`.

Добавьте код, показанный ниже, после функции `$ReplayLevel` в файле `cards.wast`:

```
(func $PlayNextLevel
  get_global $current_level
  i32.const 1
  i32.add
  call $PlayLevel
)
```

Следующий узел `func`, который нужно определить, — функция `$main`.

Функция $main

Функция `$main` не имеет параметров и не возвращает значение. Она вызывается автоматически при создании экземпляра модуля, поскольку вы указываете ее как часть узла `start`. Функция вызывает `$PlayLevel`, передавая значение `1`, чтобы начать первый уровень игры.

Добавьте код из следующего фрагмента после функции `$PlayNextLevel` в файле `cards.wast`:

```
(func $main
  i32.const 1
  call $PlayLevel
)
```

Теперь, когда вы написали все функции, определенные для базовой логики, перейдем к добавлению узлов `type`.

11.1.10. Узлы type

Как показано на рис. 11.13, в известном разделе «Тип» объявляется список всех уникальных сигнатур функций, которые будут использоваться в модуле, включая те, которые будут импортированы. Если модуль создается с помощью Binary Toolkit, то узлы s-выражения `type` необязательны, поскольку набор инструментов может определять сигнатуры на основе определений функций импорта и определенных функций в модуле. При просмотре текстового формата в инструментах разработчика браузера можно увидеть определенные s-выражения `type`, поэтому также определим их здесь для полноты картины.

Тип определяется с помощью s-выражения, которое имеет метку `type`, за которой следует необязательное имя переменной, а затем сигнатура функции. Например, следующий код определяет тип для сигнатуры функции, не имеющей параметров и не возвращающей значение:

```
(type (func))
```

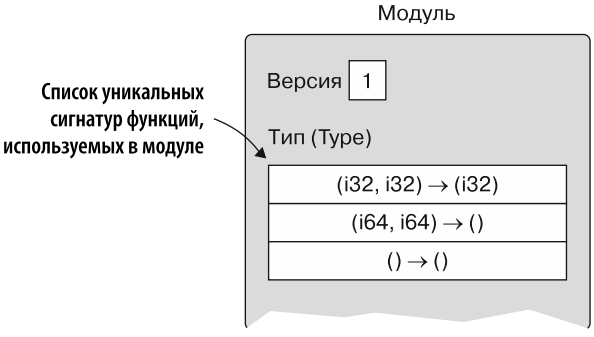


Рис. 11.13. В известном разделе «Тип» объявляется список всех уникальных сигнатур функций, которые будут использоваться в модуле, включая те, которые будут импортированы

Можно дать типу любое имя, но здесь мы будем следовать соглашению об именах Emscripten, которое представляет собой имя переменной наподобие `$FUNCSIG$vi`. Значение, следующее за вторым знаком доллара, указывает на сигнатуру функции, затем первый символ — это тип возвращаемого значения функции, а каждый дополнительный символ — тип параметра. Emscripten использует следующие символы:

- `v` — пустой тип;
- `i` — 32-разрядное целое число;
- `j` — 64-разрядное целое число;
- `f` — 32-разрядное число с плавающей запятой;
- `d` — 64-разрядное число с плавающей запятой.

Известный раздел «Тип» отображается как первый раздел в модуле, но прежде, чем его реализовать, сначала мы добавили функции модуля. Теперь можно просмотреть функции и импорты, чтобы составить список всех уникальных сигнатур функций.

Добавление узлов type в игру

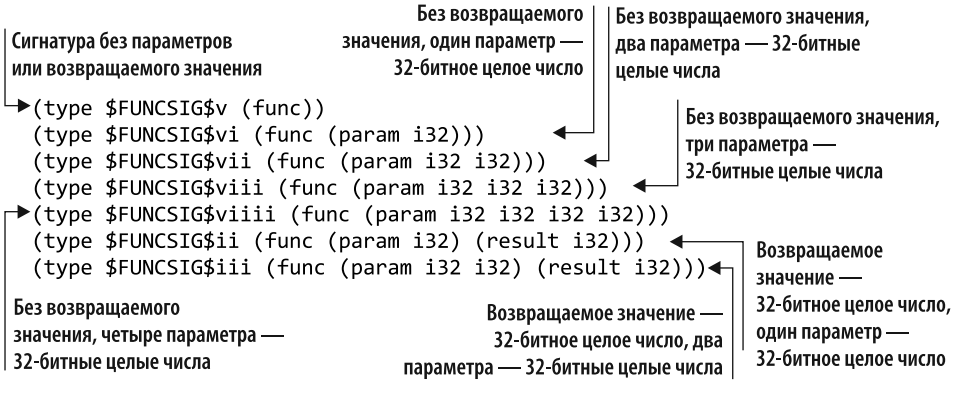
Просматривая импортированные функции и функции, созданные для этого модуля, мы собрали семь уникальных сигнатур функций, показанных в табл. 11.4.

Таблица 11.4. Семь уникальных сигнатур функций, которые используются в данном модуле

Возвращаемый тип	Параметр 1	Параметр 2	Параметр 3	Параметр 4	Сигнатура Emscripten
void	—	—	—	—	v
void	i32	—	—	—	vi
void	i32	i32	—	—	vii

Возвращаемый тип	Параметр 1	Параметр 2	Параметр 3	Параметр 4	Сигнатура Emscripten
void	i32	i32	i32	—	viii
void	i32	i32	i32	i32	viiii
i32	i32	—	—	—	ii
i32	i32	i32	—	—	iii

Получив уникальные сигнатуры функций, определенные в табл. 11.4, все, что осталось сделать, — создать узлы `type` для каждой сигнатуры. Добавьте s-выражения `type` из фрагмента, показанного ниже, в файл `cards.wast` перед узлами `import` внутри s-выражения `module`:



Последний раздел, который нужно определить для этой игры, — «Данные».

11.1.11. Узел data

Как показано на рис. 11.14, в известном разделе «Данные» объявляются данные для загрузки в линейную память модуля во время создания экземпляра.

S-выражение `data` начинается с метки `data`, за которой следует s-выражение, указывающее, где в памяти модуля должны быть данные, а за ним — строка, содержащая данные для размещения в памяти.

Строку `"SecondCardSelectedCallback"` необходимо поместить в память модуля. Этот модуль будет вручную связан с модулем, созданным Emscripten, во время выполнения, и модули, созданные Emscripten, иногда помещают собственные данные в память модуля. В результате разместим строку с индексом памяти `1024`, чтобы оставить место на случай, если модуль, сгенерированный Emscripten, тоже захочет поместить что-нибудь в память.

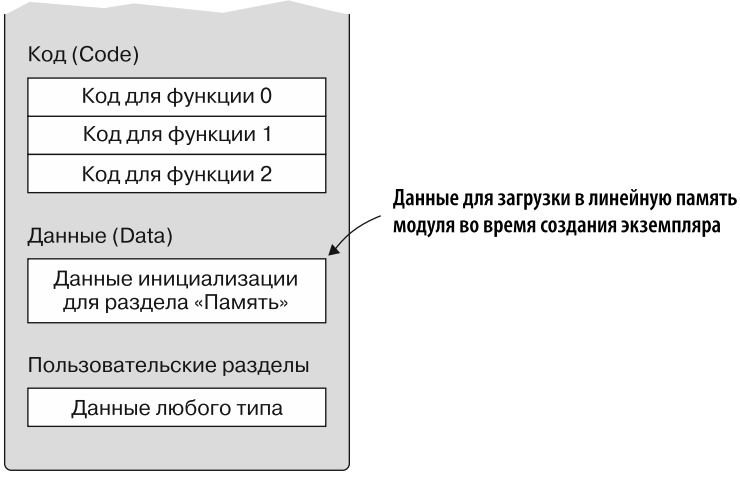


Рис. 11.14. В известном разделе «Данные» объявляются данные для загрузки в линейную память модуля во время создания экземпляра

Добавьте код, показанный ниже, в файл `cards.wast` после s-выражений `func` и s-выражение `module`, чтобы строка `"SecondCardSelectedCallback"` была помещена по индексу `1024` в памяти модуля:

```
(data (i32.const 1024) "SecondCardSelectedCallback")
```

После того как модуль текстового формата будет готов, можно сделать следующий шаг — преобразовать модуль в двоичный (рис. 11.15).

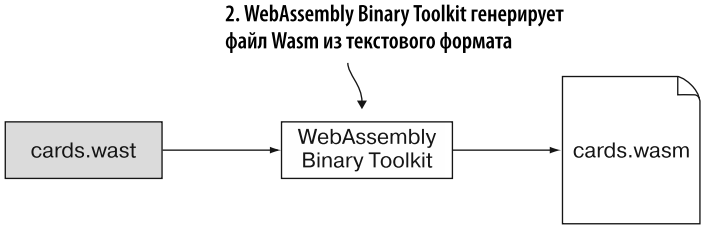


Рис. 11.15. Создание файла Wasm из текстового формата WebAssembly

11.2. СОЗДАНИЕ МОДУЛЯ WEBASSEMBLY ИЗ ТЕКСТОВОГО ФОРМАТА

Чтобы скомпилировать текстовый формат WebAssembly в модуль WebAssembly с помощью онлайн-инструмента wat2wasm, перейдите на следующий сайт: https://

webassembly.github.io/wabt/demo/wat2wasm/. Как показано на рис. 11.16, на верхней левой панели инструмента вы можете заменить существующий текст на текст из файла `cards.wast`. Инструмент автоматически создает модуль WebAssembly. Нажмите кнопку Download (Скачать), чтобы скачать сгенерированный файл WebAssembly в папку Chapter 11\source\, и назовите его `cards.wasm`.

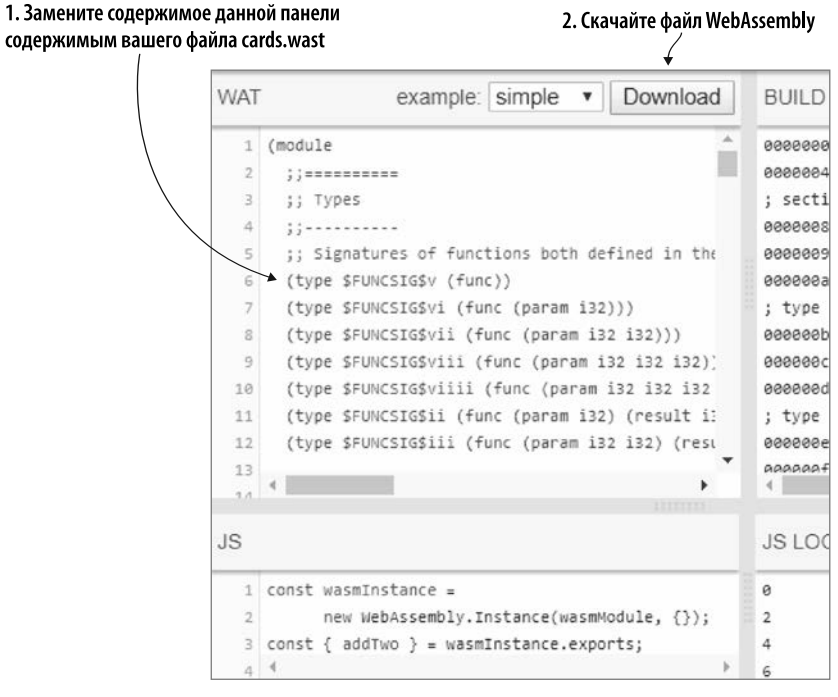


Рис. 11.16. Замените содержимое верхней левой панели содержимым вашего файла cards.wast. Затем скачайте файл WebAssembly

Сгенерировав модуль WebAssembly из кода текстового формата, можно сделать следующий шаг и создать модуль с помощью Emscripten (рис. 11.17).

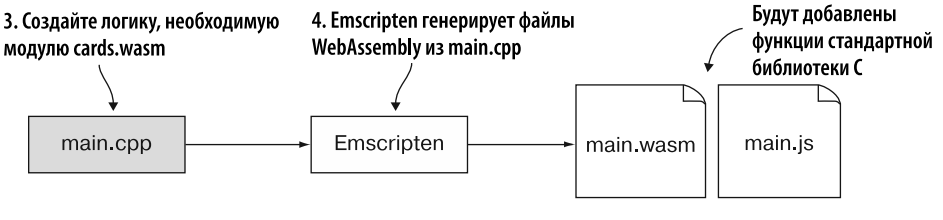


Рис. 11.17. Создание файла на C++, содержащего логику, необходимую для вашего модуля cards.wasm

11.3. МОДУЛЬ, СОЗДАННЫЙ EMSCRIPTEN

Сгенерированный Emscripten модуль предоставит вашему модулю `cards.wasm` необходимые стандартные функции библиотеки C, такие как `malloc`, `free` и функции генератора случайных чисел `srand` и `rand`. Два модуля будут связаны вручную во время выполнения. Как показано на рис. 11.18, теперь мы создадим файл на C++.

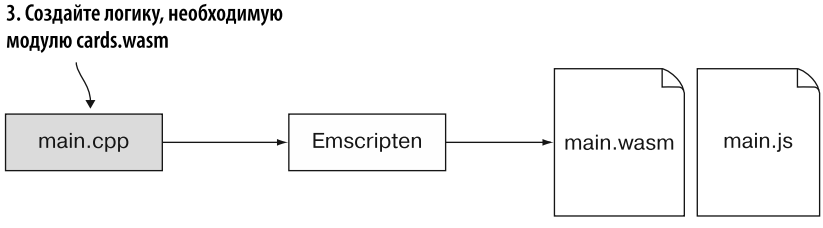


Рис. 11.18. Создание файла на C++, содержащего логику, необходимую для вашего модуля cards.wasm

11.3.1. Создание файла на C++

В папке Chapter 11\source\ создайте файл `main.cpp` и откройте его в своем любимом редакторе. Необходимо определить две функции, которые будут экспортированы для использования модулем логики игры.

Первая функция называется `SeedRandomNumberGenerator` и передает функции `srand` начальное значение. Таковым является текущее время, которое будет получено путем вызова функции `time`. Функция `time` может принимать указатель на объект `time_t` для заполнения временем, но здесь это не нужно, и потому мы просто передадим `NULL`, как показано ниже:

```
EMSCRIPTEN_KEEPALIVE
void SeedRandomNumberGenerator() { srand(time(NULL)); }
```

Вторая функция, которую нужно создать, — `GetRandomNumber`; она принимает диапазон и возвращает случайное число в этом диапазоне. Например, если значение диапазона равно 10, то случайное число будет от 0 до 9. Ниже приводится функция `GetRandomNumber`:

```
EMSCRIPTEN_KEEPALIVE
int GetRandomNumber(int range) { return (rand() % range); }
```

Модулю логики также необходим доступ к функциям `malloc` и `free`, но сгенерированный Emscripten модуль будет включать их автоматически. Добавьте код, показанный в листинге 11.12, в файл `main.cpp`.

Листинг 11.12. Содержимое файла main.cpp

```
#include <cstdlib>
#include <ctime>
#include <emscripten.h>

#ifdef __cplusplus
extern "C" {
#endif

EMSCRIPTEN_KEEPALIVE
void SeedRandomNumberGenerator() { srand(time(NULL)); }

EMSCRIPTEN_KEEPALIVE
int GetRandomNumber(int range) { return (rand() % range); }

#ifdef __cplusplus
}
#endif
```

Теперь, создав файл `main.cpp`, с помощью Emscripten превратите его в модуль WebAssembly, как показано на рис. 11.19.

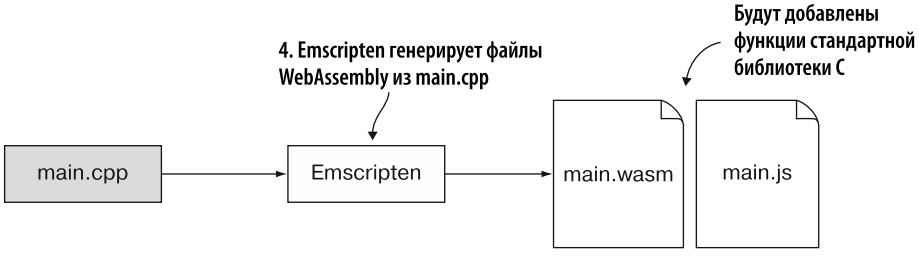


Рис. 11.19. Использование Emscripten для создания модуля WebAssembly из main.cpp

11.3.2. Создание модуля WebAssembly

Чтобы скомпилировать код в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой вы сохранили файл `main.cpp`, а затем выполните следующую команду:

```
emcc main.cpp -o main.js
```

Следующий шаг, показанный на рис. 11.20, — копирование сгенерированных файлов в определенное место, чтобы их мог использовать браузер. Затем мы создадим файлы HTML и JavaScript, необходимые для взаимодействия веб-страницы с модулями.

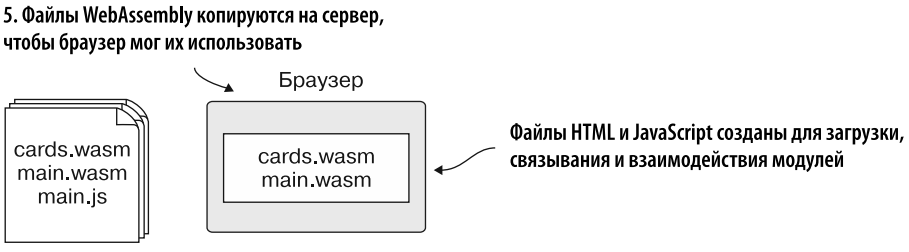


Рис. 11.20. Копирование сгенерированных файлов на сервер, чтобы браузер мог их использовать. Затем мы создадим файлы HTML и JavaScript, необходимые для взаимодействия веб-страницы с модулями

11.4. СОЗДАНИЕ ФАЙЛОВ HTML И JAVASCRIPT

В папке WebAssembly\Chapter 11\ создайте папку frontend\ для файлов, которые будут использоваться в этом разделе. Затем скопируйте следующие файлы из папки source\ в папку frontend\:

- `cards.wasm`;
- `main.wasm`;
- `main.js`;
- `editproduct.html` из папки Chapter 4\4.1 js_plumbing\frontend\; переименуйте его в `game.html`.

Создание веб-страницы игры начинается с изменения файла `game.html`.

11.4.1. Изменение файла HTML

Откройте файл `game.html` в своем любимом редакторе и измените текст в теге `title` с `Edit Product` на `Wasm Match`, как показано ниже:

```
<title>Wasm Match</title>
```

За последним тегом `script` в теге `head` добавьте следующий тег `link`, который загрузит CSS, необходимый для стилизации карт в игре:

```
<link rel="stylesheet" href="game.css">
```

ПРИМЕЧАНИЕ

Файл game.css можно найти среди исходного кода этой книги, который доступен для скачивания с сайта издателя по адресу www.manning.com/books/webassembly-in-action. Добавьте файл game.css в ту же папку, где находится и файл game.html.

Измените тег body так, чтобы он больше не имел атрибута onload="initializePage()". Тег body теперь должен выглядеть так:

```
<body>
```

После тега body измените тег div — задайте значение его атрибута class равным root-container. Затем удалите HTML в div. Теперь div должен соответствовать коду, показанному ниже:

```
<div class="root-container">
</div>
```

Класс переименован с container в root-container

HTML-код внутри div удален

В root-container div добавьте HTML-код, показанный ниже. HTML показывает название игры на веб-странице, а также текущий уровень игры. Если игрок решит перейти на следующий уровень, то JavaScript изменит тег h3, чтобы отобразить новый уровень:

```
<header class="container-fluid">
  <h1>Wasm Match</h1>
  <h3 id="currentLevel">Level 1</h3>
</header>
```

Показывает название игры на веб-странице

Отображает текущий уровень игры

После тега header, но все еще внутри root-container div добавьте тег div, показанный в следующем фрагменте кода. Карты игры будут помещены в этот div с помощью кода JavaScript:

```
<div id="cardContainer" class="container-fluid"></div>
```

Следующее, что вам нужно сделать, — добавить HTML-код, который будет представлен игроку, когда тот выиграет уровень. HTML-код укажет, какой уровень пройден, и даст возможность либо повторить текущий уровень, либо запустить следующий (если таковой существует). Добавьте HTML-код, показанный ниже, после cardContainer div и внутри root-container div:

```
  <div id="levelComplete" class="container-fluid summary"
➥ style="display:none;">
    <h1>Congratulations!</h1>
    <h3 id="levelSummary"></h3>

    <button class="btn btn-primary"
➥ onclick="replayLevel();">Replay</Button>

    <button class="btn btn-primary" id="playNextLevel"
➥ onclick="playNextLevel();">Next Level</Button>
  </div>
```

Не показывается по умолчанию. Этот div будет показан с помощью JavaScript в случае выигрыша

Содержит подробную информацию о пройденном уровне

Кнопка, которую игрок может нажать, чтобы повторить текущий уровень

Кнопка, которую игрок может нажать, чтобы перейти на следующий уровень; скрыта, если доступных уровней больше нет

Последнее, что нужно внести в файл `game.html`, — изменение значения `src` тега `script` в конце файла. Сейчас мы создадим файл `game.js`, который будет связывать два модуля и взаимодействовать с ними. Измените значение первого тега `script` на `game.js`, а значение второго тега `script` — на `main.js` (код JavaScript, созданный Emscripten):

```
<script src="game.js"></script>   ◄——— Предыдущее значение — editproduct.js
<script src="main.js"></script>   ◄——— Предыдущее значение — validate.js
```

После того как HTML будет изменен, можно сделать следующий шаг — создать JavaScript, связывающий два модуля, и наладить взаимодействие с основной логикой в модуле `cards.wasm`.

11.4.2. Создание файла JavaScript

В папке frontend\ создайте файл `game.js`, а затем откройте его в своем любимом редакторе. Добавьте глобальные переменные, показанные ниже, в файл `game.js`, чтобы хранить память модуля и экспортируемые функции:

```
let moduleMemory = null;
let moduleExports = null;
```

Следующий шаг — создание объекта `Module`, чтобы можно было обрабатывать функцию `instantiateWasm` в Emscripten. Это позволит контролировать процесс загрузки и создания экземпляра модуля WebAssembly, созданного Emscripten. После этого вы сможете загрузить и создать экземпляр файла `cards.wasm`, связав его с модулем, который сгенерирован Emscripten.

В функции `instantiateWasm` необходимо реализовать следующее:

- поместите ссылку на объект памяти `importObject` в глобальную переменную `moduleMemory` в целях дальнейшего использования вашим JavaScript;
- определите переменную, которая будет содержать экземпляр модуля `main.wasm` после того, как он будет создан;
- затем вызовите функцию `WebAssembly.instantiateStreaming`, загрузив файл `main.wasm` и передав ему `importObject`, полученный от Emscripten;
- в методе `then` объекта `instantiateStreaming Promise` определите объект `import` для модуля `cards.wasm`, передав функции из модуля `main.wasm`, а также функции JavaScript из вашего кода JavaScript. Затем вызовите `WebAssembly.instantiateStreaming` для получения модуля `cards.wasm`;
- в методе `then` промиса `instantiateStreaming` в `cards.wasm` поместите ссылку на экспорт модуля в глобальной переменной `moduleExports`. Наконец, передайте Emscripten экземпляр модуля `main.wasm`.

Добавьте код, показанный в листинге 11.13, в файл `game.js` после глобальных переменных.

Листинг 11.13. Объект Module в файле game.js

```
...

var Module = {
  instantiateWasm: function(importObject, successCallback) {
    moduleMemory = importObject.env.memory;
    let mainInstance = null;

    WebAssembly.instantiateStreaming(fetch("main.wasm"),
        importObject)
    .then(result => {
      mainInstance = result.instance;

      const sideImportObject = {
        env: {
          memory: moduleMemory,
          _malloc: mainInstance.exports._malloc,
          _free: mainInstance.exports._free,
          _SeedRandomNumberGenerator:
        ➥ mainInstance.exports._SeedRandomNumberGenerator,
          _GetRandomNumber: mainInstance.exports._GetRandomNumber,
          _GenerateCards: generateCards,
          _FlipCard: flipCard,
          _RemoveCards: removeCards,
          _LevelComplete: levelComplete,
          _Pause: pause,
        }
      };

      return WebAssembly.instantiateStreaming(fetch("cards.wasm"),
          sideImportObject)
    }).then(sideInstanceResult => {
       moduleExports = sideInstanceResult.instance.exports;

      successCallback(mainInstance);
    });

    return {};
  }
};
```

JavaScript в Emscripten будет проверять наличие этого объекта на случай, если код перегружает какую-либо логику

Позволяет контролировать создание экземпляра основного модуля

Содержит ссылку на объект памяти для использования в вашем JavaScript

Загружает и создает экземпляр модуля WebAssembly, сгенерированного с помощью Emscripten

Содержит ссылку на экземпляр модуля main.wasm

Создает объект import для модуля cards.wasm

Использует ту же память, что и экземпляр основного модуля

Загружает и создает экземпляр модуля cards.wasm

Содержит ссылку на экспорты модуля cards.wasm для использования в вашем JavaScript

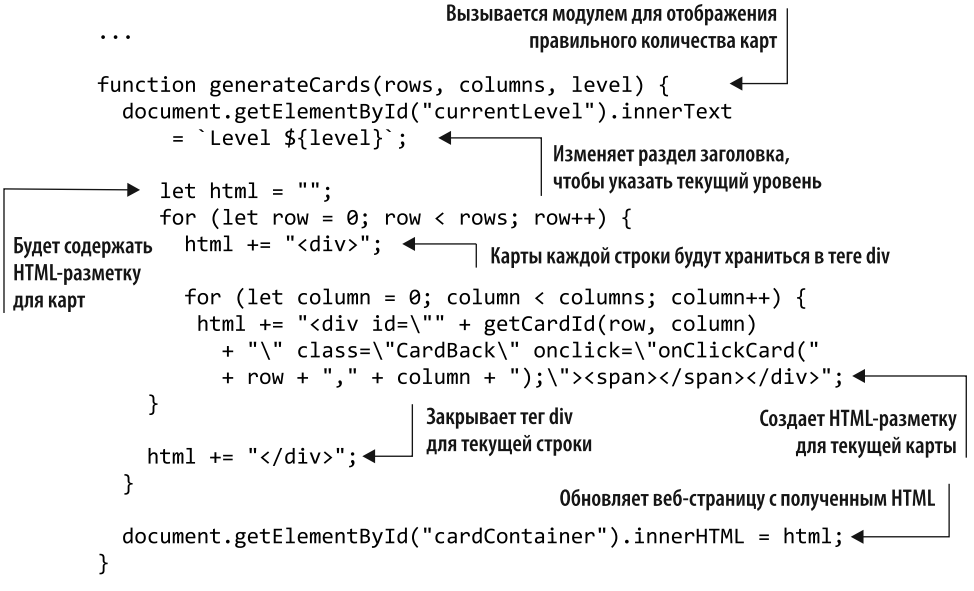
Передает экземпляр основного модуля в код JavaScript в Emscripten

Поскольку эта логика выполняется асинхронно, возвращает пустой объект

При создании экземпляра модуля `cards.wasm` автоматически запускается уровень 1 и вызывается функция `generateCards` в JavaScript, чтобы на экране отображалось нужное количество карт. Эта функция также будет вызываться, когда игрок

решит переиграть уровень или перейти на следующий. Добавьте код, показанный в листинге 11.14, в файл `game.js` после объекта `Module`.

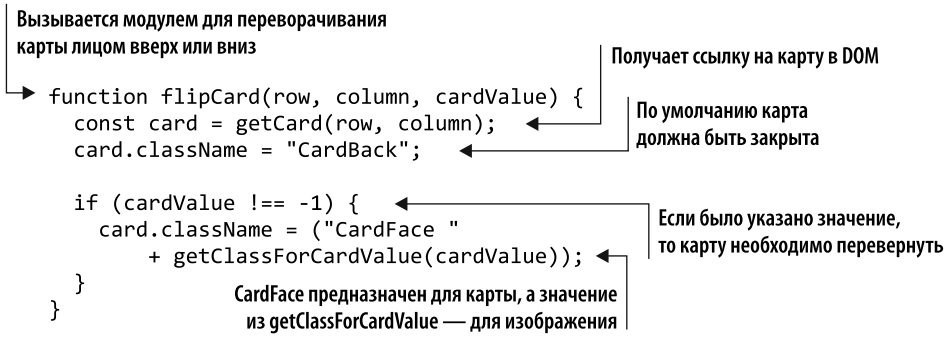
Листинг 11.14. Функция generateCards в файле game.js



Каждой отображаемой карте присваивается идентификатор на основе значений ее строки и столбца. Функция `getCardId` вернет идентификатор карты, заданный значениями строки и столбца. Добавьте функцию, показанную ниже, после функции `generateCards` в файле `game.js`:

```
function getCardId(row, column) {
  return ("card_" + row + "_" + column);
}
```

Каждый раз, когда игрок щелкает кнопкой мыши на карте, модуль вызывает функцию `flipCard`, чтобы отобразить лицевую сторону карты. Если игрок щелкает на второй карте, а карты не совпадают, то — после небольшой задержки, чтобы игрок мог видеть карты, на которых он щелкнул, — модуль снова вызовет функцию `flipCard`, чтобы обе карты перевернулись лицом вниз. Когда модулю нужно, чтобы карта была перевернута лицевой стороной вниз, он укажет `cardValue` равным `-1`. Добавьте код функции `flipCard`, показанный ниже, после функции `getCardId` в файле `game.js`:



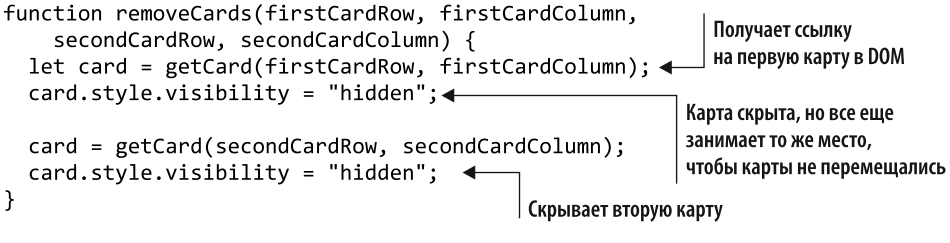
Вспомогательная функция `getCard` возвращает объект DOM для запрошенной карты на основе указанного значения строки и столбца. Добавьте функцию `getCard` после функции `flipCard` в файл `game.js`:

```
function getCard(row, column) {
  return document.getElementById(getCardId(row, column));
}
```

Когда карта открыта лицевой стороной вверх, добавляется второе имя CSS-класса, чтобы указать, какое изображение показать. В игре используются значения карт `0`, `1`, `2` и выше, в зависимости от количества уровней. Функция `getClassForCardValue` вернет имя класса, начинающееся с `Type` со значением карты, добавленным в конец (например, `Type0`). Добавьте следующий код после функции `getCard` в файл `game.js`:

```
function getClassForCardValue(cardValue) {
  return ("Type" + cardValue);
}
```

Когда игрок успешно находит две совпадающие карты, модуль вызывает функцию `removeCards`, чтобы удалить их. Добавьте код, показанный ниже, после функции `getClassForCardValue` в файле `game.js`:



Как только игрок найдет все совпадения для текущего уровня, модуль вызовет функцию `levelComplete`, чтобы JavaScript мог проинформировать игрока об этом и предложить повторить текущий уровень. Если модуль указывает, что доступен другой уровень, то игрок также получит возможность запустить следующий уровень. Добавьте код, показанный в листинге 11.15, после функции `removeCards` в файле `game.js`.

Листинг 11.15. Функция levelComplete в файле game.js

```
...

function levelComplete(level, hasAnotherLevel) {
 document.getElementById("levelComplete").style.display
➥ = "";

document.getElementById("levelSummary").innerText =
    `You've completed level ${level}!`;

if (!hasAnotherLevel) {
 document.getElementById("playNextLevel").style.display =
➥ "none";
  }
}
```

Показывает блок завершения уровня

Указывает, какой уровень игрок завершил только что

Если других уровней нет, то кнопка перехода на следующий уровень скрывается

Когда игрок щелкает кнопкой мыши на второй карте, модуль предоставляет короткую паузу, прежде чем либо перевернуть карты лицом вниз, если они не совпадают, либо скрыть их в случае их совпадения. Чтобы приостановить выполнение, модуль вызывает функцию `pause` в JavaScript, указывая функцию модуля, которую нужно вызвать из JavaScript после истечения времени ожидания. Он также передает продолжительность паузы в миллисекундах. Добавьте код, показанный ниже, в файл `game.js` после функции `levelComplete`:

```
function pause(callbackNamePointer, milliseconds) {
  window.setTimeout(function() {
    const name = ("_" +
        getStringFromMemory(callbackNamePointer));

    moduleExports[name]();
  }, milliseconds);
}
```

Создает анонимную функцию, которая будет вызываться по истечении времени ожидания

Получает имя функции из памяти модуля и ставит перед ним символ нижнего подчеркивания

Вызывает указанную функцию

Тайм-аут сработает через указанное количество миллисекунд

Функция `getStringFromMemory`, которую мы создадим следующей, копируется из кода JavaScript, с помощью которого в предыдущих главах читались строки из памяти модуля. Добавьте код, показанный в листинге 11.16, после функции `pause` в файле `game.js`.

Листинг 11.16. Функция getStringFromMemory для файла game.js

```
...

function getStringFromMemory(memoryOffset) {
  let returnValue = "";

  const size = 256;
  const bytes = new Uint8Array(moduleMemory.buffer, memoryOffset, size);

  let character = "";
  for (let i = 0; i < size; i++) {
    character = String.fromCharCode(bytes[i]);
    if (character === "\0") { break;}

    returnValue += character;
  }

  return returnValue;
}
```

Каждый раз, когда игрок щелкает кнопкой мыши на карте, тег `div` карты вызывает функцию `onClickCard`, передавая значения строки и столбца карты. Функция `onClickCard` должна передать эти значения в модуль, вызвав функцию `_CardSelected`. Добавьте код, показанный ниже, после функции `getStringFromMemory` в файле `game.js`:

```
function onClickCard(row, col) {
  moduleExports._CardSelected(row, col);
}
```

Сообщает модулю, что был щелчок на карте в этой строке и в этом столбце

Когда игрок завершает уровень, ему показывается кнопка, позволяющая повторить текущий уровень. Она вызовет функцию `replayLevel`. Функция должна будет скрыть блок завершения уровня, а затем сообщить модулю, что игрок хочет повторить уровень, вызвав функцию `_ReplayLevel`. Добавьте следующий код после функции `onClickCard` в файл `game.js`:

```
function replayLevel() {
  document.getElementById("levelComplete").style.display
➥ = "none";
  moduleExports._ReplayLevel();
}
```

Скрывает блок завершения уровня

Сообщает модулю, что игрок хочет переиграть текущий уровень

Кроме того, по завершении уровня игрок увидит кнопку, позволяющую ему перейти на следующий уровень (если такой существует). При нажатии кнопка вызовет функцию `playNextLevel` в JavaScript. Эта функция должна скрыть раздел завершения уровня, а затем сообщить модулю, что игрок хочет перейти на

следующий уровень, вызвав функцию `_PlayNextLevel`. Добавьте код из фрагмента ниже после функции `replayLevel` в файле `game.js`:

```
function playNextLevel() {
  document.getElementById("levelComplete").style.display = "none";

  moduleExports._PlayNextLevel();
}
```

Сообщает модулю, что игрок хочет перейти на следующий уровень

После того как все файлы будут созданы, можно просмотреть результаты.

11.5. ПРОСМОТР РЕЗУЛЬТАТОВ

Для просмотра результатов откройте браузер и введите `http://localhost:8080/game.html` в адресную строку, чтобы увидеть веб-страницу с игрой, показанную на рис. 11.21.

Рис. 11.21. Игра со сравнением карт выглядит так, когда игрок достигает уровня 3

Как то, что вы узнали в этой главе, можно применять в реальной ситуации?

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ В РЕАЛЬНОМ МИРЕ

Ниже приведены некоторые возможные варианты использования того, что вы узнали в этой главе.

- Как вы увидите в главе 12, текстовый формат используется браузерами при отображении содержимого модуля WebAssembly, если карты исходников недоступны. Кроме того, можно установить точку останова и пошагово выполнить код в текстовом формате, что может понадобиться для отслеживания проблемы, если вы не можете воспроизвести ее локально.
- Как вы видели в главе 6 и снова увидите в главе 12, можно включить параметр `-g` с командой `emcc`, чтобы Emscripten также выводил файл `.wast`. Если вы получаете ошибки при попытке создать экземпляр модуля или не знаете, почему что-то не работает, то иногда полезно взглянуть на содержимое этого файла.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. При использовании WebAssembly Binary Toolkit для создания модуля WebAssembly какие узлы s-выражения должны появляться перед s-выражениями `table`, `memory`, `global` и `func`?
2. Попробуйте изменить функцию `InitializeRowsAndColumns` в коде текстового формата так, чтобы теперь она поддерживала шесть уровней, а не три:
   а) уровень 4 должен состоять из трех строк и четырех столбцов;
   б) уровень 5 должен состоять из четырех строк и четырех столбцов;
   в) уровень 6 должен состоять из четырех строк и пяти столбцов.

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Существует текстовый эквивалент двоичного формата WebAssembly, который называется текстовым форматом WebAssembly. Он позволяет вам видеть модуль и работать с ним, используя понятный человеку текст, вместо того чтобы взаимодействовать напрямую с двоичным форматом.
- Текстовый формат позволяет пользователю браузера просматривать модуль WebAssembly почти так же, как он просматривает JavaScript веб-страницы.
- Текстовый формат не предназначен для написания вручную, но это все же можно сделать с помощью таких инструментов, как WebAssembly Binary Toolkit.
- В текстовом формате используются s-выражения для простого представления элементов модуля. Корневой элемент — это s-выражение `module`, а все остальные s-выражения — дочерние элементы данного узла.

- Существуют s-выражения, соответствующие известным разделам двоичного формата. Имеет значение только положение узла `import`; если он добавлен, то должен располагаться перед узлами `table`, `memory`, `global` и `func`. Кроме того, известные разделы «Функция» и «Код» в двоичном формате представлены одним функциональным s-выражением в текстовом формате.
- Четыре типа значений, поддерживаемые WebAssembly, представлены в текстовом формате как `i32` (32-битное целое число), `i64` (64-битное целое число), `f32` (32-битное число с плавающей запятой) и `f64` (64-битное число с плавающей запятой).
- Чтобы упростить работу с четырьмя типами данных, в текстовом формате добавлен объект для каждого типа с именем этого типа (например, `i32.add`).
- Код функции действует как стековая машина, в которой значения помещаются в стек и извлекаются из него. Код внутри функции может быть написан с помощью формата либо стековой машины, либо s-выражения. Браузеры отображают код функции в формате стековой машины.
- Если функция не возвращает значение, то стек должен быть пуст при выходе из нее. В противном случае элемент нужного типа должен находиться в стеке при выходе из функции.
- Вы можете ссылаться на элементы по их индексу или имени переменной.
- Параметры функции считаются локальными переменными, и их индексы ставятся перед любыми локальными переменными, определенными в функции. Кроме того, локальные переменные должны быть определены раньше всего в функции.
- В настоящее время браузеры отображают инструкции получения и установки для локальных и глобальных переменных в формате `set_local` или `get_global`. Спецификация WebAssembly была изменена с помощью нового формата: `local.set` или `global.get`, — но способ выполнения вызовов остался таким же, как и в исходном формате.

12 Отладка

В этой главе

- ✓ Различные методы отладки модулей WebAssembly.
- ✓ Обработка ошибок на этапе компиляции и во время выполнения.
- ✓ Отладка с помощью инструментов разработчика браузера.

В какой-то момент в процессе разработки вы, вероятно, обнаружите, что ваш код функционирует не так, как ожидалось, и вам понадобится найти способ отследить проблему. В одних случаях отследить неполадки настолько же просто, как прочитать код. В других придется копнуть глубже.

На момент написания статьи возможности отладки WebAssembly немного ограничены, но ситуация изменится по мере улучшения инструментов браузера и интегрированной среды разработки (integrated development environment, IDE). На данный момент доступны следующие варианты отладки модуля WebAssembly.

- Вы можете вносить небольшое количество изменений в код, а затем часто компилировать и тестировать, чтобы в случае возникновения проблемы ее было легче отследить. В этом случае чтение изменений кода может пролить свет на проблему.
- Если есть проблемы с компилятором, то вы можете дать Emscripten указание включить подробный вывод, активизировав режим отладки. В этом

режиме выводится отладочная информация и сохраняются промежуточные файлы. Переменная среды `EMCC_DEBUG` или параметр компилятора `-v` используются для управления режимом отладки. Дополнительную информацию о режиме отладки можно найти в документации Emscripten по адресу http://mng.bz/JzdZ.

- Вы можете выводить информацию из модуля в консоль браузера, используя импортированную функцию JavaScript, один из макросов Emscripten или такие функции, как `printf`. Это позволит увидеть, какие функции вызываются и какие значения хранятся в нужных переменных. При таком подходе вы начинаете с вывода областей, которые, по вашему мнению, могут содержать подсказки о проблеме. Вы можете добавлять больше записей по мере сужения радиуса поиска. (В приложении B содержится дополнительная информация о макросах Emscripten.)
- В некоторых браузерах вы можете просматривать версию модуля WebAssembly в текстовом формате, устанавливать точки останова и выполнять код по шагам. В этой главе вы узнаете, как использовать данный подход для отладки модуля.
- В Emscripten существует несколько флагов `-g` (`-g0, -g1, -g2, -g3, -g4`), которые включают все больше отладочной информации в скомпилированный вывод. Применение флага `-g` аналогично использованию `-g3`. При использовании флагов `-g` Emscripten также генерирует файл текстового формата (`.wast`) — эквивалент сгенерированного двоичного файла, что полезно при наличии проблем со связыванием, например с передачей нужных элементов в модуль во время создания экземпляра. Вы можете проверить файл текстового формата, чтобы убедиться в том, что он импортирует что-либо и были предоставлены ожидаемые элементы. Дополнительную информацию о флагах `-g` можно найти по адресу http://mng.bz/wlj5.
- Флаг `-g4` интересен тем, что генерирует карты исходников, позволяя просматривать код на C или C++ в отладчике браузера. Это многообещающий вариант отладки. Но, несмотря на то что данный подход действительно показывает ваш код на C или C++ в отладчике и использует точки останова, на момент написания отладка работает не очень хорошо. Например, если у вашей функции есть переменная параметра с определенным именем, вы не можете следить за ней, поскольку текстовый формат может фактически использовать такую переменную, как `var0`. Переход к следующему шагу в коде через отладчик может также потребовать нескольких попыток, ввиду того что скрытым образом выполняется несколько шагов текстового форматирования для одного данного оператора, и переходный вызов совершается для каждого оператора текстового формата.

В этой главе вы добавите ряд параметров отладки, которые будут использоваться при добавлении функции в игру со сравнением карт, созданную в главе 11.

12.1. РАСШИРЕНИЕ ИГРЫ

Представьте, что собираетесь расширить игру со сравнением карт таким образом, чтобы она отслеживала, сколько попыток требуется игроку для завершения уровня, как показано на рис. 12.1. Когда игрок щелкает кнопкой мыши на второй карте, это считается попыткой, независимо от того, было ли найдено совпадение.

В этой главе, чтобы узнать о доступных вариантах отладки, я предлагаю вам совершить умышленные ошибки — тем самым потребуется отладить код и определить, где и в чем проблема. Вместо того чтобы сначала вносить все эти изменения в модуль WebAssembly, а затем корректировать JavaScript, внесем изменения как в модуль, так и в JavaScript по одной функции за раз.

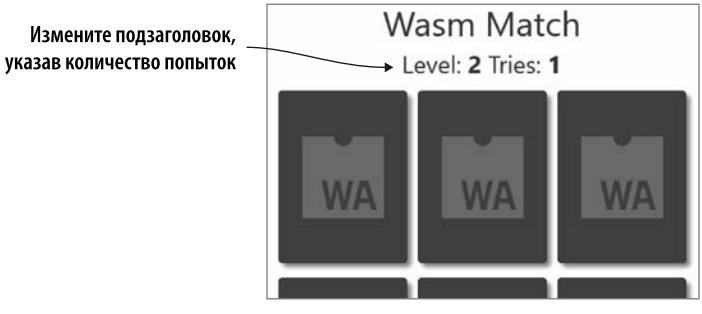


Рис. 12.1. Уровень 2 игры со сравнением карт с подзаголовком, скорректированным с учетом количества попыток

На рис. 12.2 графически представлены следующие общие шаги, с помощью которых мы будем менять игру и учитывать количество попыток.

1. Измените HTML так, чтобы подзаголовок содержал раздел для количества попыток.
2. Измените текстовый формат и JavaScript-код, чтобы отображать количество попыток на веб-странице при запуске уровня.
3. Добавьте код, чтобы увеличить количество попыток и отобразить новое значение, если игрок щелкает кнопкой мыши на второй карте.
4. Передайте количество попыток на итоговый экран, когда игрок завершит уровень.

Первый шаг — изменить HTML так, чтобы он теперь включал раздел для количества попыток.

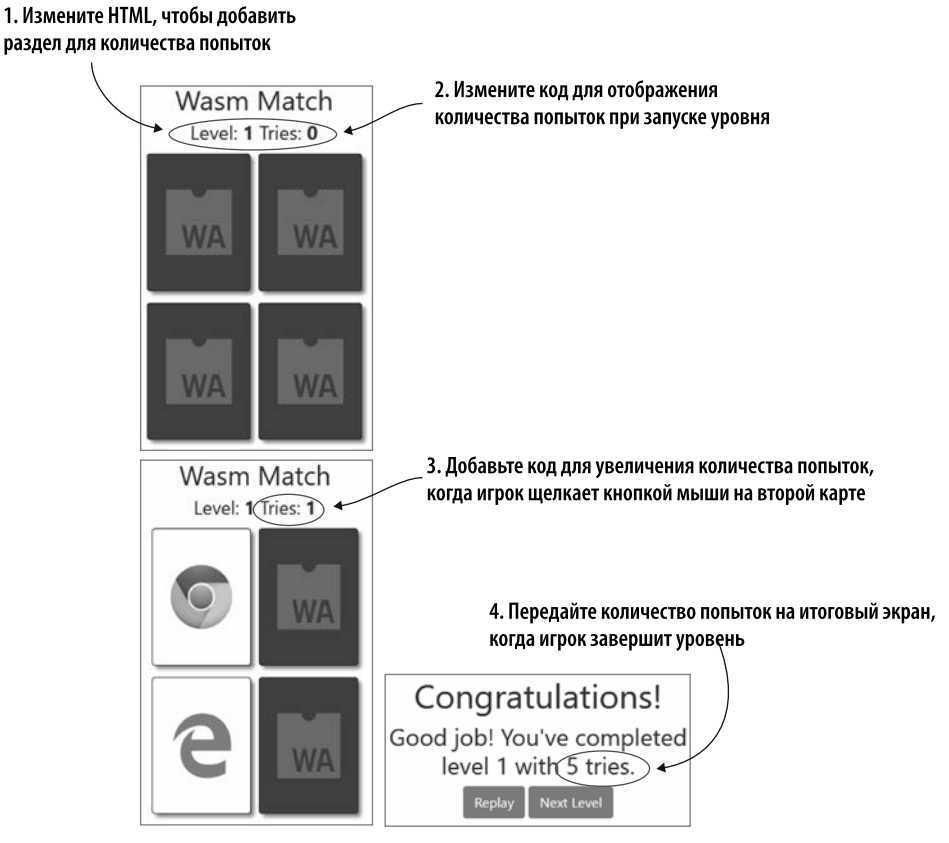


Рис. 12.2. Высокоуровневые шаги, которые будут использоваться для изменения игры и добавления количества попыток

12.2. ИЗМЕНЕНИЕ HTML

Прежде чем вы сможете внести изменения HTML, чтобы добавить количество попыток, необходимо создать папку для файлов этой главы. В папке WebAssembly\ создайте папку Chapter 12\, а затем скопируйте папки frontend\ и source\ из папки Chapter 11\.

Перейдите в папку frontend\ и откройте файл `game.html` в редакторе. В настоящий момент код JavaScript заменяет содержимое тега `h3` — тега `header` — словом `Level` (Уровень), за которым следует значение уровня (например, `Level 1`). Необходимо изменить тег `h3` так, чтобы он включал и количество попыток:

- удалите атрибут `id` и его значение из тега `h3`;

- добавьте текст `Level:`, а затем тег `span` с атрибутом `id`, содержащим значение `currentLevel` (`id="currentLevel"`). Этот диапазон теперь будет хранить текущий уровень;
- добавьте текст `Tries:`, а затем тег `span` с атрибутом `id`, содержащим значение `tries` (`id="tries"`). В этом диапазоне будет отображаться количество попыток.

Тег заголовка в файле `game.html` теперь должен соответствовать коду, показанному ниже:

```
<header class="container-fluid">
  <h1>Wasm Match</h1>
  <h3>
    Level: <span id="currentLevel">1</span>
    Tries: <span id="tries"></span>
  </h3>
</header>
```

Удаляем атрибут id

После того как HTML будет изменен, можно сделать следующий шаг — изменить текстовый формат WebAssembly и код JavaScript, чтобы отображалось значение количества попыток при запуске уровня.

12.3. ОТОБРАЖЕНИЕ КОЛИЧЕСТВА ПОПЫТОК

В следующей части процесса вам нужно изменить код так, чтобы показывать количество попыток при запуске уровня. Для этого мы сделаем ряд шагов, также изображенных на рис. 12.3.

1. Измените функцию `generateCards` в JavaScript, чтобы получить новый параметр, указывающий количество попыток, отображаемых при запуске уровня.
2. В текстовом формате создайте глобальную переменную `$tries` для хранения количества попыток. Затем измените функцию `$PlayLevel`, чтобы передать количество попыток функции `generateCards` в JavaScript.
3. Используйте набор инструментов WebAssembly Binary Toolkit для создания модуля WebAssembly из текстового формата (`card.wasm`).
4. Скопируйте сгенерированный файл WebAssembly на сервер, чтобы браузер мог использовать этот файл, а затем проверьте, работают ли изменения должным образом.

Первый элемент, который нужно изменить, — функция `generateCards` в файле `game.js`.

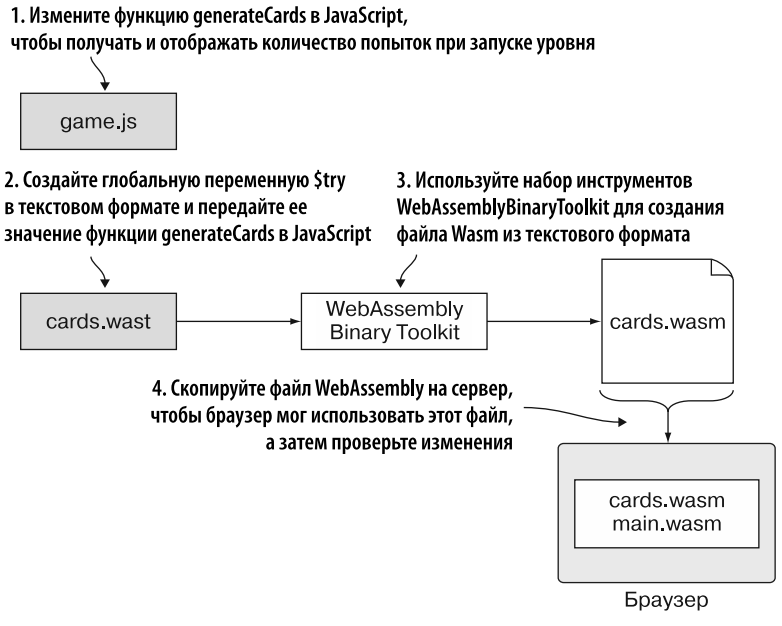


Рис. 12.3. Измените код JavaScript и текстового формата, чтобы отображалось количество попыток при запуске уровня

12.3.1. Функция generateCards в JavaScript

Откройте файл `game.js` и найдите функцию `generateCards`. Вам нужно добавить четвертый параметр `tries` в функцию после существующих параметров. Этот параметр будет передан функции модулем WebAssembly, чтобы его можно было отобразить на веб-странице при запуске уровня.

Измените функцию `generateCards` в файле `game.js` так, чтобы она соответствовала коду, показанному в листинге 12.1.

Листинг 12.1. Функция generateCards в файле game.js

```
...

function generateCards(rows, columns, level, tries) {
  document.getElementById("currentLevel").innerText = level;
  document.getElementById("tries").innerText = tries;

  let html = "";
  for (let row = 0; row < rows; row++) {
    html += "<div>";
```

Добавляет параметр tries

Просто передает само значение уровня

Добавьте эту строку кода, чтобы обновить элемент с количеством попыток

```
    for (let column = 0; column < columns; column++) {
      html += "<div id=\"" + getCardId(row, column)
          + "\" class=\"CardBack\" onclick=\"onClickCard("
          + row + "," + column + ");\"><span></span></div>";
    }

    html += "</div>";
  }

  document.getElementById("cardContainer").innerHTML = html;
}
...
```

Как показано на рис. 12.4, следующее, что нужно сделать, — создать глобальную переменную `$tries` в текстовом формате для хранения количества попыток, сделанных игроком. Затем следует передать это значение в функцию `generateCards` в JavaScript.

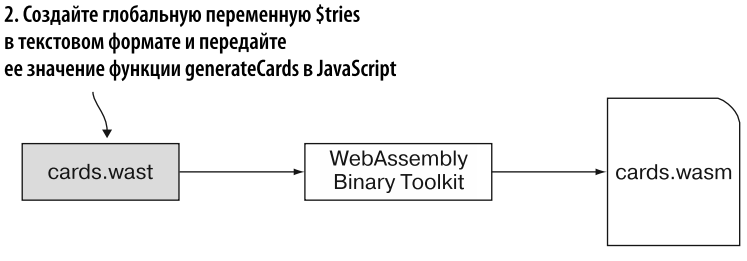


Рис. 12.4. Создайте глобальную переменную $tries в коде текстового формата и передайте ее значение функции JavaScript generateCards

12.3.2. Изменения в коде текстового формата

В этом подразделе мы создадим глобальную переменную `$tries` и передадим ее в функцию `generateCards` в JavaScript. Откройте файл `cards.wast`, а затем перейдите в известный раздел «Глобальные переменные».

Добавьте изменяемую глобальную переменную `i32` с именем `$tries` после глобальной переменной `$matches_remaining` в файле `cards.wast`. Глобальная переменная должна выглядеть следующим образом:

```
(global $tries (mut i32) (i32.const 0))
```

Определив глобальную переменную, нужно передать ее в качестве четвертого параметра функции `generateCards` в JavaScript. Перейдите к функции `$PlayLevel` и поместите значение `$tries` в стек как четвертый параметр для вызова функции `$GenerateCards` (между переменной `$level` и строкой кода вызова `$GenerateCards`).

В файле `cards.wast` измененная функция `$PlayLevel` должна теперь выглядеть так:

```
(func $PlayLevel (param $level i32)
  get_local $level
  call $InitializeCards

  get_global $rows
  get_global $columns
  get_local $level
  get_global $tries
  call $GenerateCards
)
```

Значение tries помещается в стек в качестве четвертого параметра generateCard

В конце функции `$InitializeCards` после строки кода `call $ShuffleArray` в файле `cards.wast` добавьте следующий код для сброса значения `$tries` каждый раз при запуске уровня:

```
get_global 6
set_global $tries
```

Код текстового формата изменен. На рис. 12.5 показан следующий шаг, в котором мы будем использовать WebAssembly Binary Toolkit, чтобы преобразовывать код текстового формата в файл `cards.wasm`.

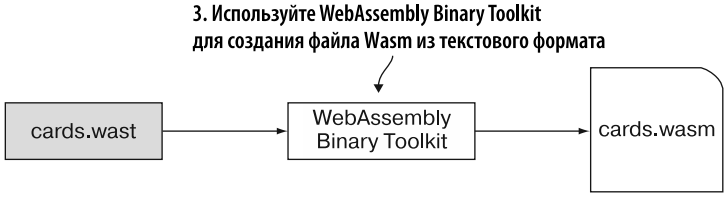


Рис. 12.5. Используйте WebAssembly Binary Toolkit для создания файла cards.wasm из текстового формата

12.3.3. Создание файла Wasm

Чтобы скомпилировать текстовый формат WebAssembly в модуль WebAssembly с помощью онлайн-инструмента wat2wasm, перейдите на сайт https://webassembly.github.io/wabt/demo/wat2wasm/ и скопируйте содержимое файла `cards.wast` в верхнюю левую панель инструмента. К сожалению, вы увидите сообщение об ошибке в правом верхнем углу инструмента, как показано на рис. 12.6.

Ниже приводится полное сообщение об ошибке:

```
test.wast:329:5: error: type mismatch in function, expected [] but got [i32]
call $GenerateCards
```

Сообщение об ошибке указывает на вызов $GenerateCards

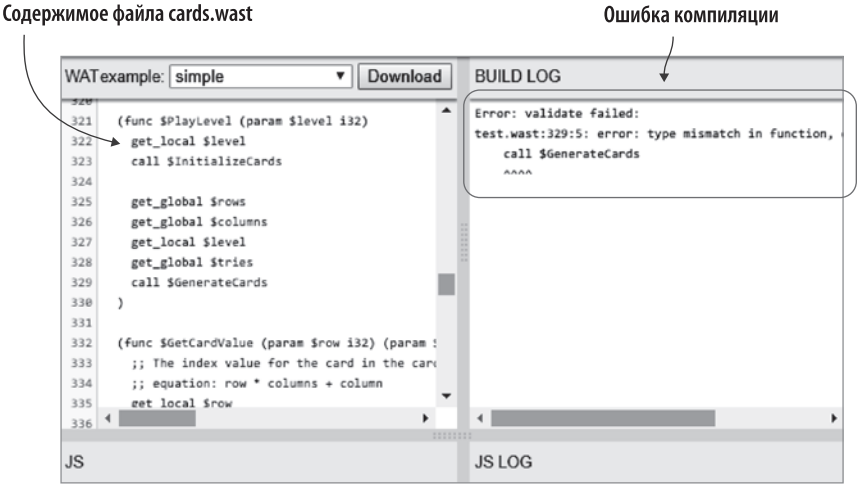


Рис. 12.6. Ошибка компиляции содержимого файла cards.wast

Поскольку файл `cards.wast` компилировался без проблем в главе 11 и в сообщении об ошибке упоминается функция `$GenerateCards`, ошибка, вероятно, как-то связана с изменением, внесенным в функцию `$PlayLevel`. Просмотрите код на наличие экземпляров строки `$GenerateCards` — вы наверняка обнаружите, что пошло не так. В известном разделе «Импорт» был добавлен узел `import` для функции `_GenerateCards` JavaScript, но вы не добавили четвертый параметр `i32` в сигнатуру функции.

Посмотрите на функцию `$PlayLevel`, показанную в следующем фрагменте кода, — она по-прежнему считает, что функции `$GenerateCards` нужны три параметра. В результате три верхних элемента в стеке будут извлечены и переданы в функцию `$GenerateCards`. Это оставит значение `$rows` в стеке. При возврате функции `$GenerateCards` функция `$PlayLevel` заканчивается тем, что некий объект еще находится в стеке. Функция `$PlayLevel` не должна ничего возвращать, поэтому наличие чего-либо в стеке вызывает ошибку:

```
(func $PlayLevel (param $level i32)
  get_local $level
  call $InitializeCards

  get_global $rows
  get_global $columns
  get_local $level
  get_global $tries
  call $GenerateCards
)
```

Не возвращает значения. По завершении функции стек должен быть пустым

Помещается в стек первым. Останется в стеке при вызове $GenerateCards

Три верхних элемента извлекаются из стека и передаются в $GenerateCards

Чтобы решить эту проблему, перейдите в известный раздел «Импорт» в файле `cards.wast` и добавьте четвертый параметр `i32` в функцию `$GenerateCards`, как показано ниже:

```
(import "env" "_GenerateCards"
  (func $GenerateCards (param i32 i32 i32 i32))
)
```

Скопируйте и вставьте содержимое файла `cards.wast` на верхнюю левую панель инструмента wat2wasm еще раз, а затем загрузите новый файл Wasm в папку frontend\.

Теперь, получив новый файл `cards.wasm`, перейдем к следующему шагу — тестированию изменений, показанному на рис. 12.7.

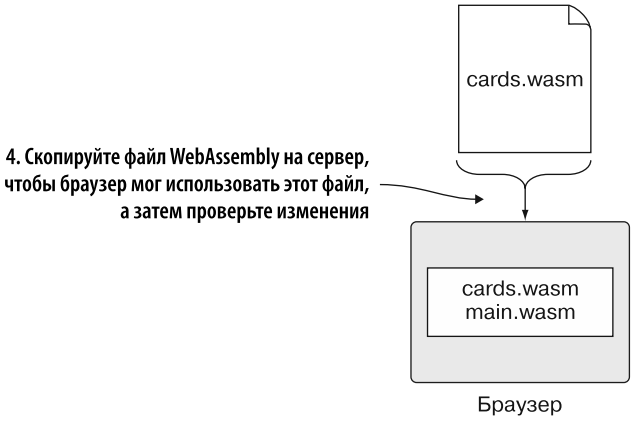


Рис. 12.7. Скопируйте файл cards.wasm для использования в браузере, а затем проверьте свои изменения

12.3.4. Тестирование изменений

Изменяя файл `games.html`, вы не помещали значение количества попыток в тег `span`; это означает, что если изменения не сработают, то при запуске уровня на сайте будет отображаться только текст `Tries:` (Попытки:). Если внесенные изменения сработали, то при запуске уровня вы увидите текст `Tries: 0`. Откройте браузер и введите `http://localhost:8080/game.html` в адресную строку, чтобы увидеть измененную веб-страницу, показанную на рис. 12.8.

На рис. 12.9 показан следующий шаг, необходимый для реализации логики количества попыток. Когда игрок щелкает кнопкой мыши на второй карте, глобальная переменная `$tries` увеличится, а веб-страница обновится с новым значением.

Рис. 12.8. Внесенные изменения работают, поскольку значение 0 отображается рядом с меткой Tries

Рис. 12.9. Количество попыток увеличивается, когда игрок щелкает кнопкой мыши на второй карте

12.4. УВЕЛИЧЕНИЕ КОЛИЧЕСТВА ПОПЫТОК

В следующей части процесса нужно увеличить количество попыток, когда игрок щелкает кнопкой мыши на второй карте. Для этого будут использоваться следующие шаги, которые также показаны на рис. 12.10.

1. Добавьте в файл `game.js` функцию JavaScript (`updateTriesTotal`), которая будет получать значение попыток от модуля и обновлять веб-страницу с полученным значением.
2. Измените текстовый формат для импорта функции `updateTriesTotal` в JavaScript. Дайте текстовому формату указание увеличивать значение `$tries`, когда игрок щелкает кнопкой мыши на второй карте, а затем передайте это значение функции JavaScript.
3. Используйте WebAssembly Binary Toolkit для создания модуля WebAssembly из текстового формата (`card.wasm`).
4. Скопируйте сгенерированный файл WebAssembly на сервер, чтобы браузер мог использовать этот файл, а затем проверьте, работают ли изменения должным образом.

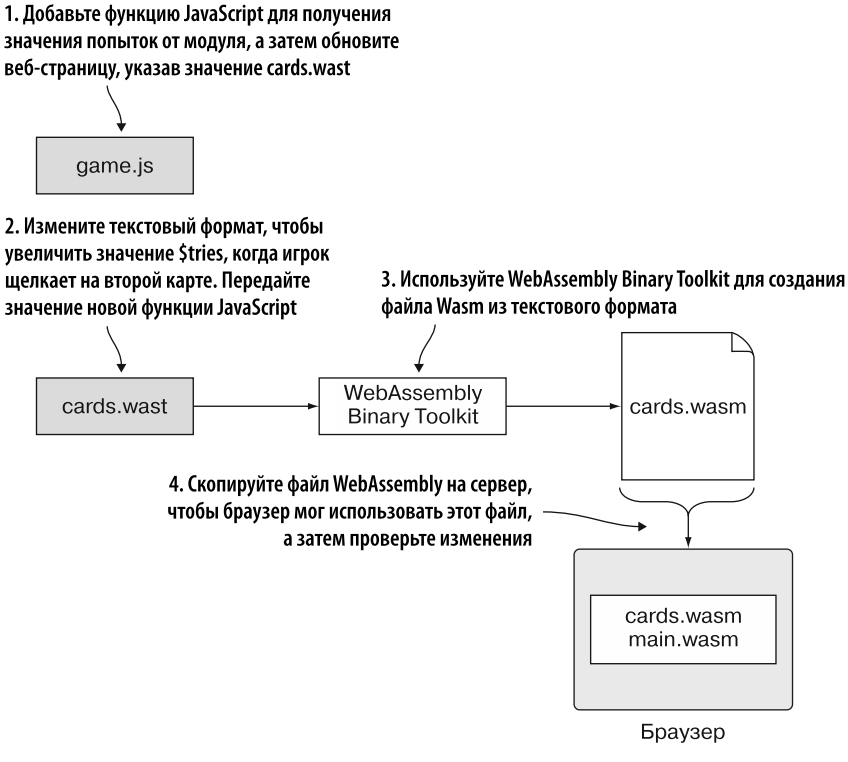


Рис. 12.10. Увеличение количества попыток, когда игрок щелкает на второй карте

Первый шаг — создание функции `updateTriesTotal` в файле `game.js`.

12.4.1. Функция updateTriesTotal в JavaScript

В файле `game.js` создайте функцию `updateTriesTotal`, которая получает параметр `tries` и обновляет веб-страницу его значением. Поместите функцию после функции `generateCards`, а затем скопируйте строку кода `document.getElementById` для значения `tries` из функции `generateCards` в функцию `updateTriesTotal`.

Функция `updateTriesTotal` в файле `game.js` должна выглядеть следующим образом:

```
function updateTriesTotal(tries) {
  document.getElementById("tries").innerText = tries;
}
```

В функции `generateCards` файла `game.js` замените строку кода `document.getElementById` для значения `tries` с помощью вызова функции `updateTriesTotal`:

```
updateTriesTotal(tries);
```

После того как JavaScript-код будет изменен, можно сделать следующий шаг, показанный на рис. 12.11, и изменить код текстового формата, увеличивая значение `$tries`, когда игрок щелкает кнопкой мыши на второй карте. Затем новое значение `$tries` будет передано новой функции JavaScript.

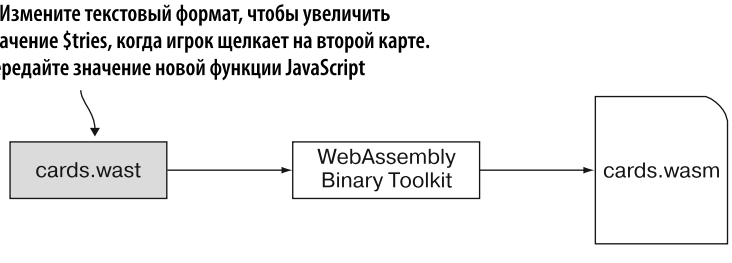


Рис. 12.11. Текстовый формат увеличивает значение $tries, когда игрок щелкает на второй карте. Затем значение будет передано новой функции JavaScript

12.4.2. Изменение текстового формата

Необходимо добавить узел `import` для функции `updateTriesTotal` в JavaScript, чтобы можно было передать обновленное значение `$tries` в код JavaScript и отобразить его на веб-странице. В файле `cards.wast` перейдите в известный раздел «Импорт» и добавьте узел `import` для функции `$UpdateTriesTotal`,

которая получает один параметр `i32`. Поместите узел `import` после узла `import` в `$GenerateCards`.

Получившийся узел `import` в файле `cards.wast` должен выглядеть так:

```
(import "env" "_UpdateTriesTotal"
  (func $UpdateTriesTotal (param i32))
)
```

Перейдите к функции `$SecondCardSelectedCallback`. Она вызывается после короткой паузы, когда игрок щелкает на второй карте, чтобы он мог видеть карту до того, как она будет удалена или перевернута лицевой стороной вниз, в зависимости от того, совпадают ли карты.

После оператора `if` увеличьте глобальную переменную `$tries` на единицу. Затем передайте значение `$tries` в функцию `$UpdateTriesTotal`, чтобы код JavaScript обновил веб-страницу новым значением.

Код, показанный в листинге 12.2, отображает изменения, внесенные в функцию `$SecondCardSelectedCallback` в файле `cards.wast`. Часть кода функции в листинге была опущена, чтобы сфокусироваться только на изменениях.

Листинг 12.2. Функция $SecondCardSelectedCallback в файле cards.wast

```
(func $SecondCardSelectedCallback
  (local $is_last_level i32)

  get_global $first_card_value
  get_global $second_card_value
  i32.eq
  if

  else

  end

  get_global $tries
  i32.const 10
  i32.add
  set_global $tries

  get_global $tries
  call $UpdateTriesTotal

  Остальная часть функции
)
```

Карты совпадают

Карты не совпадают

Увеличивает значение

JavaScript дано указание удалить карты. Значение $matches_remaining уменьшается на 1

JavaScript дано указание перевернуть карты рубашкой вверх

Передает значение в JavaScript, чтобы веб-страницу можно было обновить

После того как код текстового формата будет изменен, вы можете сгенерировать файл WebAssembly из текстового формата, как показано на рис. 12.12.

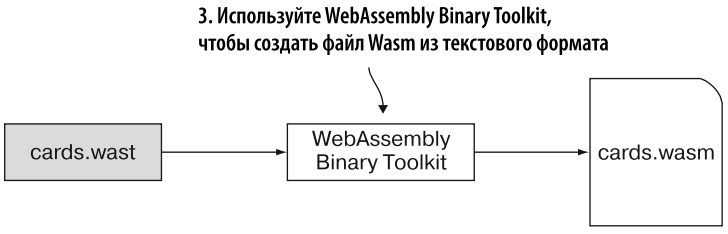


Рис. 12.12. Чтобы создать файл WebAssembly, воспользуемся WebAssembly Binary Toolkit

12.4.3. Создание файла Wasm

Чтобы скомпилировать текстовый формат WebAssembly в модуль WebAssembly с помощью онлайн-инструмента wat2wasm, перейдите на сайт https://webassembly.github.io/wabt/demo/wat2wasm/. Вставьте содержимое файла `cards.wast` на левую верхнюю панель инструмента, как показано на рис. 12.13. Затем нажмите кнопку Download (Скачать), чтобы скачать файл WebAssembly в папку frontend\, и присвойте файлу имя `cards.wasm`.

Рис. 12.13. Вставьте содержимое файла cards.wast на верхнюю левую панель инструмента, а затем скачайте файл WebAssembly, назвав его cards.wasm

Создав новый файл `cards.wasm`, перейдите к следующему шагу, показанному на рис. 12.14, — к тестированию изменений.

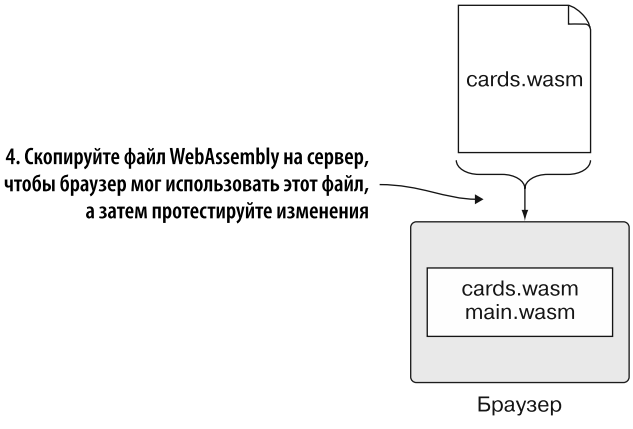


Рис. 12.14. Скопируйте файл cards.wasm на сервер и протестируйте свои изменения

12.4.4. Тестирование изменений

С учетом изменений, внесенных в JavaScript и текстовый формат, после щелчка кнопкой мыши на карте значение `$tries` увеличится на 1, а затем это же значение будет обновлено на веб-странице. Откройте браузер и введите `http://localhost:8080/game.html` в адресную строку, чтобы убедиться, что измененный код работает так, как ожидается. К сожалению, как показано на рис. 12.15, что-то не работает — ваша игра не отображается.

Рис. 12.15. Что-то не работает — значение не отображается

Когда веб-страница ведет себя не так, как ожидается, — например, не отображается должным образом или не реагирует на щелчки кнопкой мыши, — проблема может быть связана с ошибкой в JavaScript. Нажмите F12, чтобы открыть в браузере

инструменты разработчика, а затем просмотрите консоль, чтобы узнать, есть ли сообщения об ошибках. Оказывается, как показано на рис. 12.16, в консоли есть ошибка JavaScript, которая говорит про поле `_UpdateTriesTotal`.

Рис. 12.16. Появилась ошибка JavaScript, говорящая о проблеме с полем _UpdateTriesTotal

Благодаря рис. 12.16 мы узнаем два полезных факта, первый из которых — слово *LinkError*. Так называется ошибка, возникающая при появлении проблемы с созданием экземпляра модуля в WebAssembly. Более подробную информацию об ошибках LinkError можно найти на странице MDN Web Docs по адресу http://mng.bz/qXjx.

Другой полезный факт заключается в том, что ошибка неким образом связана с полем `_UpdateTriesTotal`. Таково имя функции, которое вы дали узлу `import` для импорта функции JavaScript, как показано в следующем фрагменте кода, написанном ранее:

```
(import "env" "_UpdateTriesTotal"
  (func $UpdateTriesTotal (param i32))
)
```

С виду узел `import` в коде текстового формата кажется правильным. Модуль также скомпилировался без проблем, поэтому затруднение, похоже, не в самом модуле. В таком случае нужно взглянуть на JavaScript.

Откройте файл `game.js`. Функция JavaScript `updateTriesTotal`, показанная в следующем фрагменте кода, имеет правильную сигнатуру (принимает один параметр и не возвращает значения), поэтому сама функция выглядит правильно:

```
function updateTriesTotal(tries) {
  document.getElementById("tries").innerText = tries;
}
```

Поскольку мы получили LinkError и эта ошибка связана с файлом `cards.wasm`, то обратите внимание на раздел кода `WebAssembly.instantiateStreaming` для

`cards.wasm`. Если вы посмотрите на `sideImportObject`, то заметите, что свойство `_UpdateTriesTotal` не было добавлено.

В файле `game.js` измените `sideImportObject` так, чтобы добавить свойство `_UpdateTriesTotal` для функции `updateTriesTotal`. Поместите его после свойства `_GenerateCards`, как показано в листинге 12.3.

Листинг 12.3. SideImportObject в файле game.js

```
const sideImportObject = {
  env: {
    memory: moduleMemory,
    _malloc: mainInstance.exports._malloc,
    _free: mainInstance.exports._free,
    _SeedRandomNumberGenerator:
        mainInstance.exports._SeedRandomNumberGenerator,
    _GetRandomNumber: mainInstance.exports._GetRandomNumber,
    _GenerateCards: generateCards,
    _UpdateTriesTotal: updateTriesTotal,   <-- Передает функцию updateTriesTotal модулю
    _FlipCard: flipCard,
    _RemoveCards: removeCards,
    _LevelComplete: levelComplete,
    _Pause: pause,
  }
};
```

Сохраните файл `game.js`, а затем обновите веб-страницу — вы должны увидеть, что ошибка JavaScript исчезла и страница отображается должным образом.

Если щелкнуть кнопкой мыши на двух картах, после того как они были перевернуты лицевой стороной вниз или удалены, то можно заметить, что на веб-странице обновится значение попыток. К сожалению, как показано на рис. 12.17, что-то не так, поскольку количество попыток увеличивается с шагом 10.

Чтобы решить эту проблему, мы выполним по шагам код текущего текстового формата в браузере. Если вы используете Firefox, то можете пропустить следующий пункт и просмотреть текст, озаглавленный «Отладка в Firefox».

Отладка в Chrome

Как показано на рис. 12.18, чтобы просмотреть содержимое модулей WebAssembly в Chrome, нужно нажать `F12` для открытия инструментов разработчика, а затем перейти ко вкладке Sources (Источники). В разделе wasm на левой панели модули отображаются в том порядке, в котором были загружены. В данном случае первый модуль — `main.wasm`, а второй — `card.wasm`.

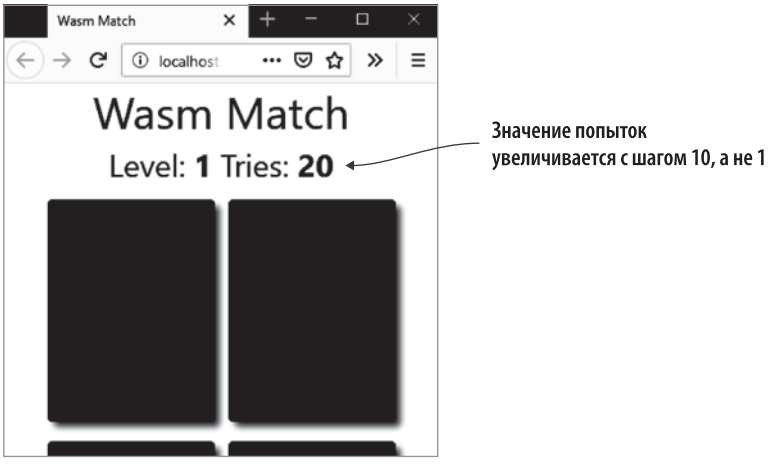


Рис. 12.17. Значение попыток показывает наличие проблемы

СОВЕТ

Иногда при первом открытии инструментов разработчика раздел wasm не отображается. Обновите веб-страницу, и он должен загрузиться.

Развернув модуль WebAssembly, вы увидите список всех встроенных функций модуля, обозначенных индексом, отсчитываемым от нуля. Импортированные функции не показаны, но их индексы находятся перед индексами встроенных функций, поэтому индексы, показанные на рис. 12.18, начинаются с 10, а не с 0.

Щелкнув кнопкой мыши на функции, вы можете увидеть ее в текстовом формате на правой панели. Затем вы можете щелкнуть на номере строки на правой панели, чтобы установить точку останова. Задав ее, просто запустите нужный раздел кода на веб-странице — код остановится в этой точке, позволяя вам шагать по нему построчно, чтобы увидеть, что происходит.

В текстовом формате вы можете вызывать функции и переменные по их индексу или использовать имя переменной. В инструментах разработчика Chrome применяются индексы, а не имена переменных. Это может сбивать с толку, и потому полезно открыть одновременно исходный код и текстовый формат, чтобы сравнить и понять, на какую функцию вы сейчас смотрите.

Если вы используете браузер Chrome, то можете пропустить следующий пункт, в котором показаны инструменты разработчика Firefox при отладке модуля WebAssembly.

Отладка модуля WebAssembly с помощью инструментов разработчика в Chrome

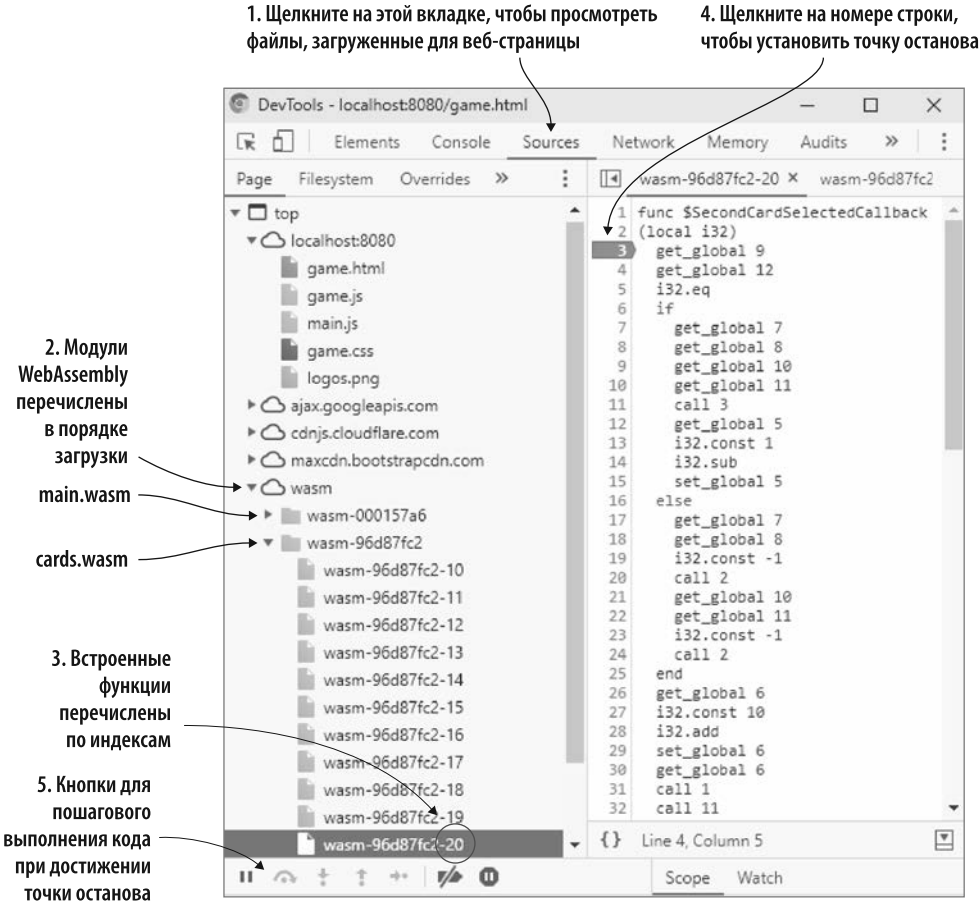


Рис. 12.18. Инструменты разработчика Chrome для отладки модуля WebAssembly

Отладка в Firefox

Как показано на рис. 12.19, чтобы просмотреть содержимое модулей WebAssembly в Firefox, вам нужно нажать F12 для открытия инструментов разработчика и затем перейти на вкладку Debugger (Отладчик). На левой панели щелкните на интересующем вас файле WebAssembly; текстовый формат этого файла будет показан на правой панели.

Затем вы можете щелкнуть на номере строки на правой панели, чтобы установить точку останова. После этого вам просто нужно запустить данный раздел кода на веб-странице — код остановится в указанной точке, позволяя вам пройти его по шагам, чтобы увидеть происходящее.

Если посмотреть на функцию на рис. 12.19, то приведенные имена переменных не очень удобны. Если код ссылается на локальную переменную, то она либо является параметром, либо определена в начале функции, поэтому не так сложно определить, что представляет собой значение. С другой стороны, глобальные переменные определяются в начале файла, вследствие чего такие переменные, как `$global7` и `$global12`, понять труднее. Задачу можно упростить: для этого полезно одновременно открыть исходный код или текстовый формат, чтобы можно было сравнивать просматриваемый материал.

Чтобы определить причину проблемы с увеличением значения `$tries` на 10, а не на 1, необходимо выполнить отладку функции `$SecondCardSelectedCallback`.

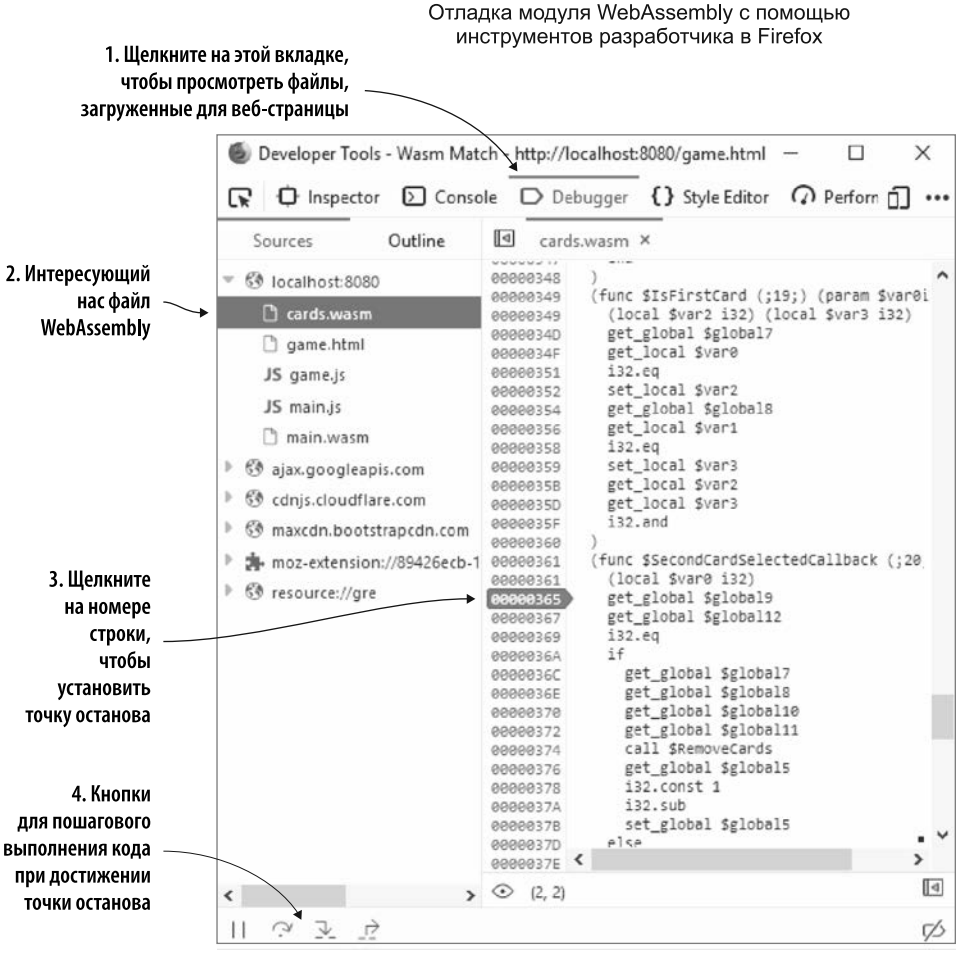


Рис. 12.19. Инструменты разработчика Firefox для отладки модуля WebAssembly

Отладка функции $SecondCardSelectedCallback

Прежде чем начать отладку функции `$SecondCardSelectedCallback`, полезно знать, что представляет каждый индекс глобальной переменной, поскольку и Firefox, и Chrome ссылаются на глобальные переменные по их индексу в коде функции. Если посмотреть на известный раздел «Глобальные переменные» вашего файла `cards.wast`, то глобальные переменные и их индексы будут соответствовать перечисленным в табл. 12.1.

Таблица 12.1. Глобальные переменные и их соответствующие индексы

Глобальная переменная	Индекс
$MAX_LEVEL	0
$cards	1
$current_level	2
$rows	3
$columns	4
$matches_remaining	5
$tries	6
$first_card_row	7
$first_card_column	8
$first_card_value	9
$second_card_row	10
$second_card_column	11
$second_card_value	12
$execution_paused	13

В инструментах разработчика браузера перейдите к функции `$SecondCard-SelectedCallback` и установите точку останова в первой строке кода `get_global` после объявления локальной переменной. В оставшейся части этого подраздела мы будем использовать инструменты разработчика в Firefox.

Чтобы сработала точка останова, щелкните на двух картах. Как показано на рис. 12.20, одна из панелей в окне отладчика — это Scopes (Области видимости). Если вы развернете разделы Block (Блок), то обнаружите, что в одном из них отображаются значения глобальных переменных для области действия данной функции. Первые два вызова `get_global` в функции предназначены для `global9` и `global12`, которые, согласно табл. 12.1, содержат значения первой и второй карты соответственно. Значения глобальных переменных могут отличаться от того, что вы видите в инструментах разработчика вашего браузера, поскольку карты

отсортированы случайным образом. В нашем случае значения `global9` и `global12` содержат `1` и `0` соответственно.

Просмотр значений глобальных переменных

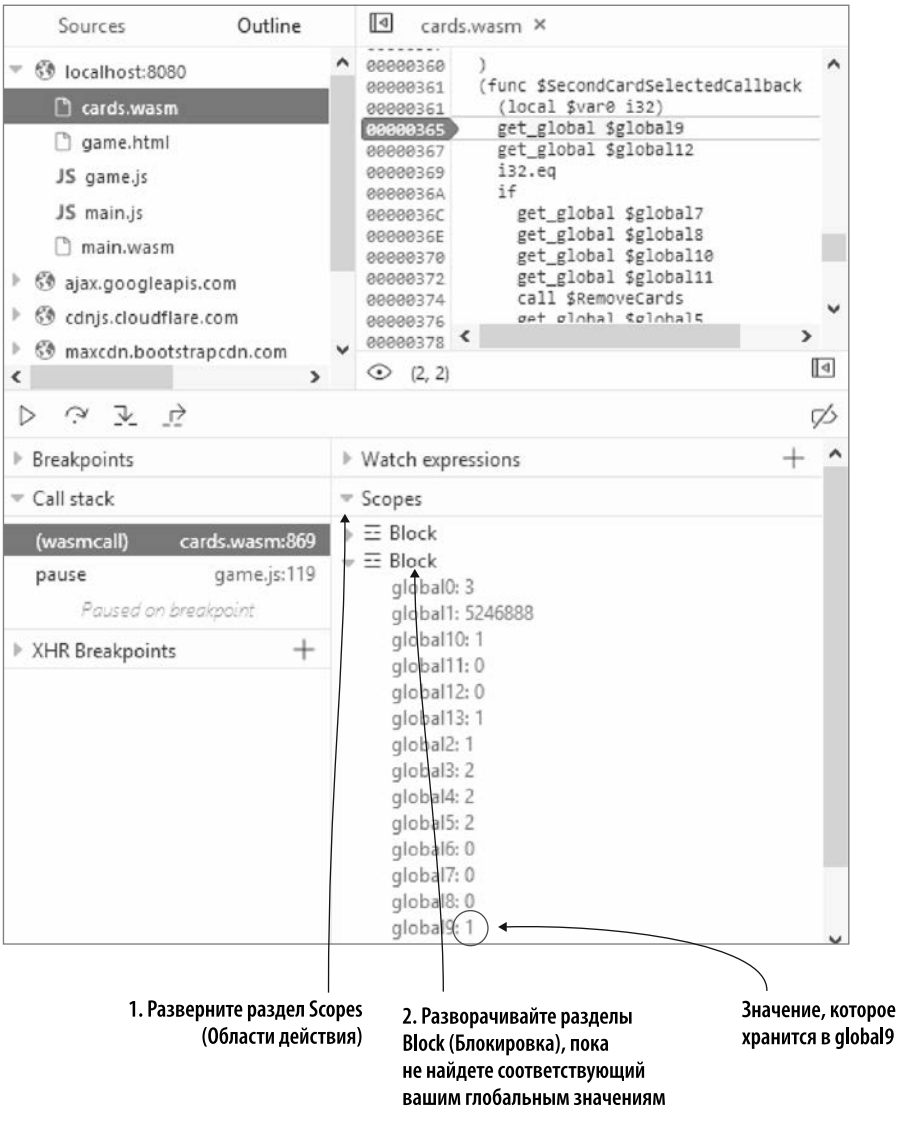


Рис. 12.20. Раздел Scopes (Области видимости) в Firefox, показывающий глобальные переменные в области видимости этой функции

СПРАВКА

В инструментах разработчика Chrome на панели Scopes (Области видимости) не отображаются значения глобальных переменных. Если вы развернете локальный элемент на данной панели, то появится элемент стека, который показывает значения, которые находятся в стеке в настоящее время. Firefox не показывает, что находится в стеке. В зависимости от потребностей в отладке вам может понадобиться использовать инструменты отладки одного браузера в одних случаях и отладчик другого браузера — в других.

Значения `global9` (в данном случае `1`) и `global12` (в данном случае `0`) помещаются в стек, после чего вызывается `i32.eq`. Вызов `i32.eq` извлекает два верхних значения из стека, сравнивает их, а затем помещает в стек значение, указывающее, равны ли они. Затем оператор `if` извлекает верхний элемент из стека и входит в блок `if`, если значение истинно. Если оно ложно и есть условие `else`, то код войдет в условие `else`. В этом случае два глобальных значения не равны, и потому код входит в условие `else`.

Код в условии `else` помещает значения из `global7` и `global8` (значения строки и столбца первой выбранной карты соответственно) в стек вместе со значением `-1`. Затем он вызывает функцию `FlipCards` в JavaScript. `-1` дает функции `FlipCards` указание перевернуть карту лицевой стороной вниз. `FlipCards` снова вызывается со значениями `global10` и `global11`, чтобы вторая карта была перевернута лицом вниз.

После оператора `if` `global6` (счетчик `$tries`) помещается в стек вместе со значением `i32.const`, равным `10`. Значения `global6` и `i32.const 10` извлекаются из стека вызовом `i32.add`, суммируются, а затем результат помещается обратно в стек, откуда затем помещается в переменную `global6`.

Оказывается, проблема с увеличением значения попыток на 10, а не на 1 — это опечатка: в коде использовалось `i32.const 10`, а не `i32.const 1`. Найдите в файле `cards.wast` функцию `$SecondCardSelectedCallback`. Измените код, увеличивающий значение `$tries`, чтобы он использовал `i32.const 1`, а не `10`, как показано ниже:

```
get_global $tries
i32.const 1  <-------- Измените с 10 на 1
i32.add
set_global $tries
```

Повторная генерация файла Wasm

Для того чтобы скомпилировать текстовый формат WebAssembly в модуль WebAssembly, вставьте содержимое файла `cards.wast` на левую верхнюю панель

онлайн-инструмента wat2wasm: https://webassembly.github.io/wabt/demo/wat2wasm/. Нажмите кнопку Download (Загрузить), чтобы загрузить файл WebAssembly в папку frontend\, и назовите файл `cards.wasm`. Обновите веб-страницу, чтобы убедиться, что щелчки кнопкой мыши на двух картах теперь увеличивают количество попыток на 1, а не на 10.

Теперь количество попыток обновляется каждый раз, когда игрок щелкает кнопкой мыши на второй карте, — пора выполнить последний шаг. Как показано на рис. 12.21, количество попыток будет передано на экран итогов, когда уровень будет завершен.

Рис. 12.21. Количество попыток будет передано на итоговый экран, когда игрок завершит уровень

12.5. ОБНОВЛЕНИЕ ЭКРАНА ИТОГОВ

Следующая часть процесса требует обновить поздравительное сообщение, указав количество попыток. Для этого будем использовать ряд шагов, изображенных на рис. 12.22.

1. Обновите функцию `levelComplete` в JavaScript, чтобы принять другой параметр для количества попыток. Затем измените текст на экране итогов, добавив количество попыток.
2. Измените текстовый формат, чтобы передать значение `$tries` в функцию `levelComplete` в JavaScript.
3. Используйте набор инструментов WebAssembly Binary Toolkit, чтобы создать модуль WebAssembly из текстового формата (`cards.wasm`).
4. Скопируйте сгенерированный файл WebAssembly на сервер, чтобы браузер мог использовать этот файл, а затем проверьте, работают ли изменения должным образом.

Первый шаг — изменить функцию `levelComplete` в файле `game.js`.

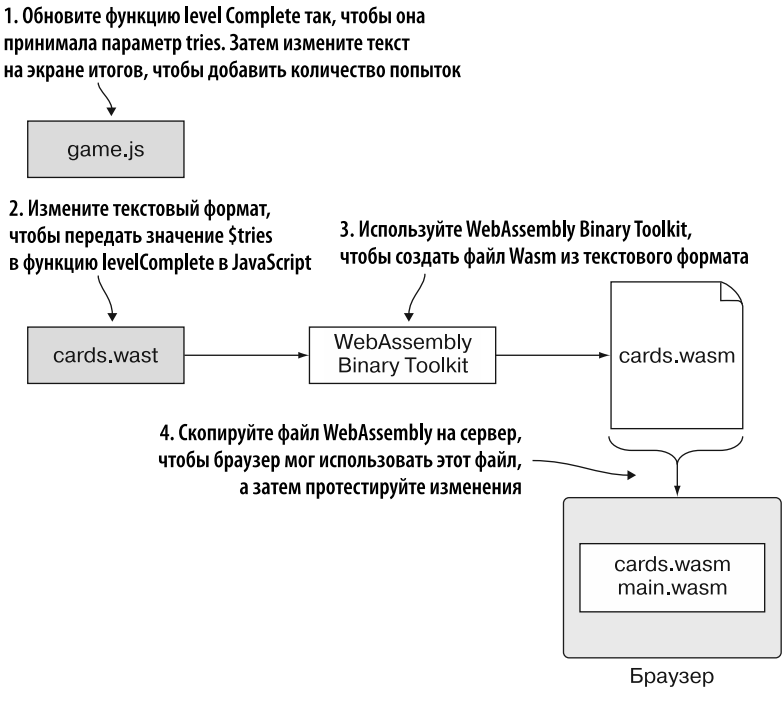


Рис. 12.22. Шаги по добавлению количества попыток в поздравительное сообщение на экране итогов

12.5.1. Функция levelComplete в JavaScript

В файле `game.js` измените функцию `levelComplete` так, чтобы в качестве второго параметра между `level` и `hasAnotherLevel` передавался параметр `tries`. Затем скорректируйте текст, переданный в элемент DOM `levelSummary`, так, чтобы он включал в себя количество попыток. Функция `levelComplete` в файле `game.js` должна соответствовать коду, показанному в листинге 12.4.

Листинг 12.4. Функция levelComplete в файле game.js

```
function levelComplete(level, tries, hasAnotherLevel) {
  document.getElementById("levelComplete").style.display = "";
  document.getElementById("levelSummary").innerText = `Good job!
➥ You've completed level ${level} with ${tries} tries.`;

  if (!hasAnotherLevel) {
    document.getElementById("playNextLevel").style.display = "none";
  }
}
```

Добавлен параметр tries

Текст изменен так, чтобы показывать количество попыток

После того как изменения в JavaScript будут внесены, можно сделать следующий шаг — изменить текстовый формат для передачи значения `$tries` в `levelComplete` (рис. 12.23).

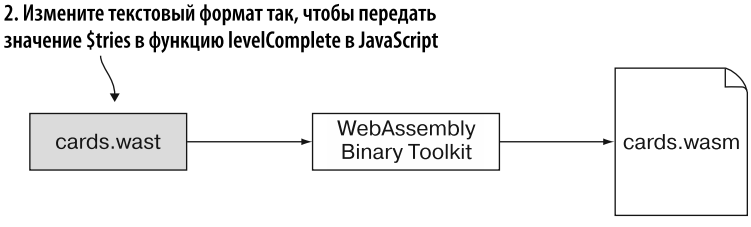


Рис. 12.23. Передайте значение $tries в функцию levelComplete в JavaScript

12.5.2. Изменение текстового формата

В коде текстового формата необходимо изменить логику так, чтобы значение `$tries` передавалось в функцию `levelComplete` в JavaScript. Однако прежде, чем изменять вызов `levelComplete`, необходимо настроить сигнатуру узла `import` для этой функции так, чтобы она принимала три параметра `i32`.

В файле `cards.wast` найдите узел `import` для функции `levelComplete` в JavaScript и добавьте третий параметр `i32`. Измененный узел `import` теперь должен соответствовать коду, показанному ниже:

```
(import "env" "_LevelComplete"
  (func $LevelComplete (param i32 i32 i32))
)
```

Функция `$LevelComplete` вызывается в конце функции `$SecondCardSelectedCallback`, поэтому перейдите к данной функции в файле `cards.wast`. Ожидается, что значение `$tries` будет вторым параметром `levelComplete`, вследствие этого поместите вызов `get_global` для значения `$tries` между вызовом `get_global` для `$current_level` и вызовом `get_local` для значений `$is_last_level`.

В файле `cards.wast` вызов функции `$LevelComplete` теперь должен выглядеть так:

```
get_global $current_level
get_global $tries   <-- Помещает значение из $tries в стек
get_local $is_last_level
call $LevelComplete
```

После того как код текстового формата будет изменен, можно сгенерировать файл WebAssembly из текстового формата, как показано на рис. 12.24.

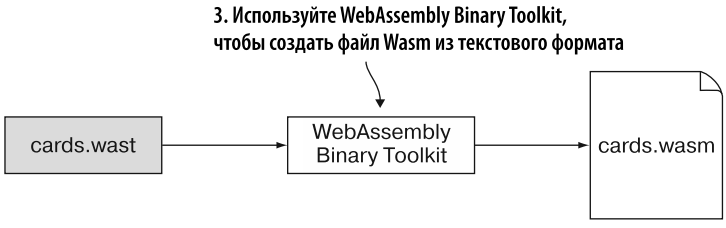


Рис. 12.24. Создание файла WebAssembly из текстового формата

12.5.3. Создание файла Wasm

Чтобы скомпилировать содержимое файла `card.wast` в модуль WebAssembly с помощью онлайн-инструмента wat2wasm, перейдите на сайт https://webassembly.github.io/wabt/demo/wat2wasm/. Вставьте содержимое файла `cards.wast` на верхнюю левую панель инструмента, как показано на рис. 12.25. Нажмите кнопку Download (Скачать) и скачайте файл WebAssembly в папку frontend\. Назовите его `cards.wasm`.

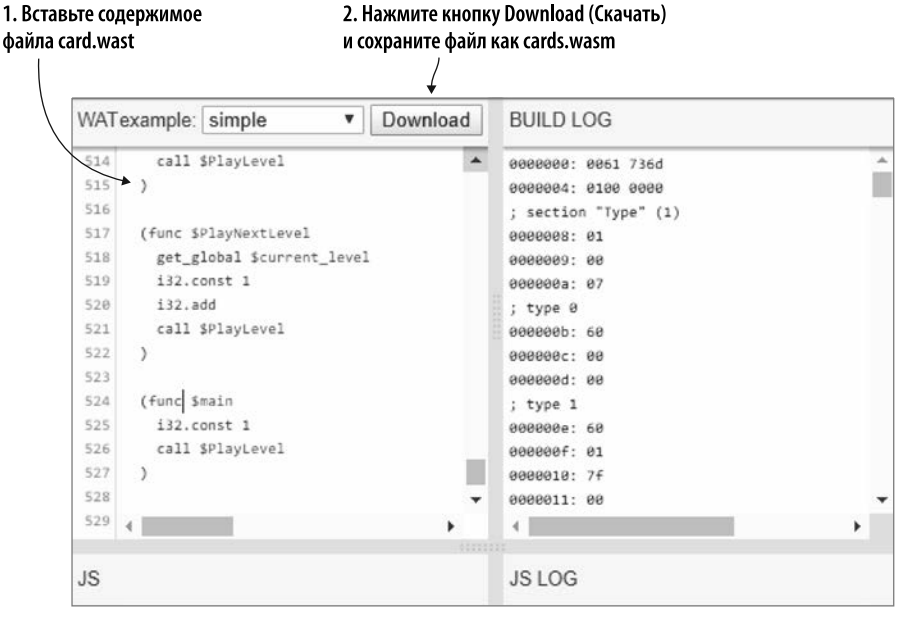


Рис. 12.25. Вставьте содержимое файла cards.wast на верхнюю левую панель, а затем скачайте файл WebAssembly. Назовите его cards.wasm

Получив новый файл `cards.wasm`, можно сделать следующий шаг, показанный на рис. 12.26, — протестировать изменения.

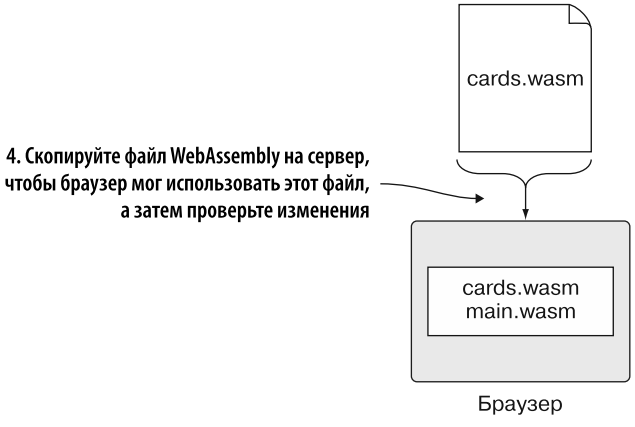


Рис. 12.26. Скопируйте файл cards.wasm на сервер, а затем проверьте изменения

12.5.4. Тестирование изменений

Чтобы проверить, правильно ли работают внесенные вами изменения, откройте браузер и введите `http://localhost:8080/game.html` в адресную строку. Когда вы пройдете уровень, на экране итогов будет отображаться количество попыток, как показано на рис. 12.27.

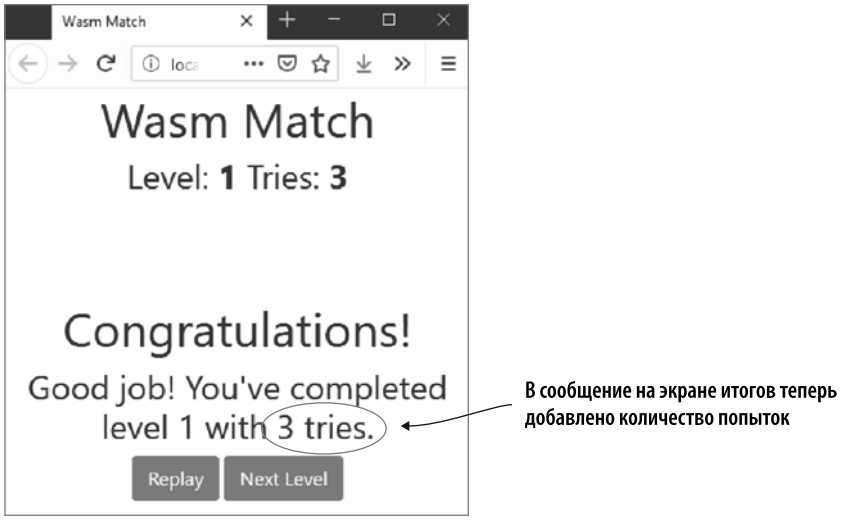


Рис. 12.27. Экран итогов с добавленным количеством попыток

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какими двумя способами можно получить доступ к переменной или вызвать функцию?
2. Возможно, вы заметили, что значение попыток не сбрасывается, когда вы переигрываете уровень или переходите на следующий. Используйте логирование в консоль браузера, чтобы определить источник проблемы.

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Emscripten предлагает переменную среды `EMCC_DEBUG` и флаг `-v` для управления режимом отладки. Если этот параметр включен, то выводится отладочная информация и сохраняются промежуточные файлы.
- В Emscripten также есть несколько флагов `-g`, чтобы предоставлять больше отладочной информации в скомпилированном файле. В дополнение к расширенной отладочной информации Emscripten также выдает версию в текстовом формате (`.wast`), эквивалентную сгенерированному двоичному файлу, что может быть полезно при поиске источника проблем.
- Логирование информации в консоль браузера — один из способов отладить то, что происходит в модуле.
- Флаг `-g4` можно использовать для того, чтобы дать Emscripten указание генерировать карты исходников — тогда код на C или C++ можно будет просматривать в браузере. На момент написания этой книги данная функция все еще требует доработки в браузерах.
- В некоторых браузерах можно просмотреть версию загруженного двоичного файла в текстовом формате. Вы можете установить точки останова, выполнить код и, в зависимости от браузера, просмотреть либо значения переменных, либо значения, находящиеся в стеке.
- В настоящее время функции отладки браузеров неодинаковы для разных браузеров, поэтому может понадобиться переключаться между браузерами в зависимости от того, что требует отладки.

13 Тестирование и все, что с ним связано

В этой главе

- ✓ Создание автоматических тестов с использованием Mocha.
- ✓ Запуск тестов из командной строки в Node.js.
- ✓ Запуск тестов в браузерах, которые вы собираетесь поддерживать.

Во время разработки проекта наступает момент, когда нужно протестировать некий элемент, чтобы убедиться в его корректной работе. Выполнение тестов вручную в начале проекта может показаться достаточным, но по мере того, как количество кода растет, этапы тестирования должны становиться более подробными, чтобы не допустить ошибки. Проблема в том, что тестирование становится утомительным: как бы вы ни старались, легко что-то упустить, а проблема может ускользнуть.

Вдобавок при ручном тестировании вы зависите от своего тестировщика. Иногда тестировщики могут проверять лишь одну часть кода за раз, и они могут работать только с определенной скоростью, иначе начнут делать ошибки.

При работе с продуктом, который должен поддерживать несколько платформ, тестирование становится еще более сложным, поскольку всякий раз при внесении изменений в код вам необходимо повторять одни и те же тесты на каждой поддерживаемой платформе.

Автоматическое тестирование требует предварительного создания тестов, но если вы их написали, то получаете ряд преимуществ.

- Тесты, в зависимости от типа, могут выполняться быстро.
- Их можно запускать сколько угодно раз. Например, вы можете запустить их перед выпуском нового кода с целью убедиться, что внесенное изменение не повлияло на другие элементы системы.
- Тесты можно запускать в любое время. Например, есть возможность запланировать выполнение более длительных тестов в ночное время и просматривать результаты утром, по возвращении к работе.
- Каждый раз тесты будут работать одинаково.
- Можно запускать одни и те же тесты на разных платформах. Это полезно при написании модулей WebAssembly для браузеров, поскольку нужно убедиться, что модули работают должным образом в нескольких браузерах.

Автоматические тесты не устраняют необходимость в ручном тестировании, но могут обрабатывать однообразные элементы, позволяя вам сосредоточиться на других областях.

При разработке вы можете реализовать несколько различных типов тестирования.

- *Модульные* тесты пишутся для тестирования отдельных модулей (например, функций), чтобы разработчик мог убедиться в корректной работе логики. Эти тесты разработаны как быстрые, поскольку пишутся таким образом, чтобы тестируемый код не зависел от вещей наподобие файловой системы, базы данных или веб-запросов.

  Модульные тесты настоятельно рекомендуется использовать, поскольку они позволяют обнаруживать ошибки на ранних этапах процесса разработки. Они также помогают быстро выявлять проблемы регрессии, если вы вносите изменения, влияющие на другие области.

- *Интеграционные* тесты подтверждают, что две или более области работают вместе ожидаемым образом. В этом случае выполнение тестов может занять больше времени, поскольку они могут иметь внешние зависимости от таких вещей, как база данных или файловая система.

- Существует много других типов тестирования. Так, *приемочные тесты* позволяют убедиться, что система удовлетворяет бизнес-требованиям, а *тесты производительности* проверяют, что система адекватно работает при большой нагрузке. На сайте https://en.wikipedia.org/wiki/Software_testing есть дополнительная информация о различных доступных типах тестирования программного обеспечения.

Предположим, вы написали модуль WebAssembly и теперь хотите создать несколько тестов, чтобы убедиться в нормальной работе функций. Вы хотите использовать среду JavaScript, которая позволяет запускать тесты из командной строки, чтобы получить подтверждение, что все работает по мере написания кода. Но то, что работает в одном браузере, может не работать точно таким же образом в другом. В некоторых случаях функция из одного браузера не существует в другом, поэтому вам также понадобится платформа JavaScript, которая позволит запускать тесты в браузере.

В этой главе вы узнаете, как писать автоматические интеграционные тесты, которые позволяют быстро и легко убедиться, что модули WebAssembly работают должным образом. Вы также узнаете, как запускать эти тесты в поддерживаемых браузерах.

Данная глава представляет обзор того, как можно тестировать модули WebAssembly, но не содержит обзор различных доступных платформ или подробное описание выбранной среды.

СПРАВКА

Доступно множество фреймворков для тестирования JavaScript, среди которых наиболее популярными являются Jest, Mocha и Puppeteer. Несколько фреймворков перечислены в статье Нвосе Лотанна *Top Javascript Testing Frameworks in Demand for 2019* на http://mng.bz/py1w (Medium). В этой книге в учебных целях мы будем использовать Mocha.

Первое, что нужно сделать, — установить среду тестирования JavaScript.

13.1. УСТАНОВКА СРЕДЫ ТЕСТИРОВАНИЯ JAVASCRIPT

Эта глава предъявляет два требования к среде тестирования:

- тесты должны запускаться из вашей среды разработки (IDE) или из командной строки, чтобы вы могли быстро проверить, все ли работает должным образом, прежде чем выпускать новый код;
- тесты также необходимо запускать в браузере, чтобы вы могли убедиться, что все работает должным образом в браузерах, которые вы собираетесь поддерживать.

Основываясь на этих двух требованиях, я выбрал для данной главы фреймворк Mocha, который запускается на Node.js из командной строки, а также может работать в браузере. (Узнать больше о Mocha можно на https://mochajs.org.)

Если вы планируете использовать Mocha только в среде Node.js, то можете применить встроенный модуль Node.js `assert` в качестве *библиотеки утверждений* — это инструмент, который проверяет, соответствует ли результат теста ожидаемому. Например, ниже показано, как вызывается тестируемый код, а затем библиотека утверждений используется для проверки того, что результат равен `2`:

```
const result = codeUnderTest.increment(1);
expect(result).to.equal(2);
```

Библиотека утверждений также выполняет проверку таким образом, чтобы было легче читать и поддерживать ее вывод по сравнению с набором операторов `if`, генерирующих исключения, как в данном примере:

```
const result = codeUnderTest.increment(1);
if (result !== 2) {
  throw new Error(`expected 2 but received ${result}`);
}
```

В этой главе, поскольку вы будете запускать тесты как в Node.js, так и в браузере, я выбрал библиотеку Chai для единообразия, ввиду того что ее можно использовать и там и там. У Chai также есть несколько стилей утверждения, что позволяет вам применять самый удобный стиль. В данной главе мы обратимся к стилю `Expect`, но вы можете использовать и стиль `Assert`, поскольку он совместим с браузером и очень похож на модуль `assert` в Node.js. Более подробную информацию о стилях утверждения, доступных в Chai, можно найти на сайте www.chaijs.com/api.

СПРАВКА

Хотя Chai была выбрана в качестве библиотеки утверждений для этой главы, вместе с Mocha вы можете использовать любую библиотеку утверждений. Список нескольких доступных библиотек можно найти на https://mochajs.org/#assertions.

Как уже упоминалось, фреймворк Mocha работает на Node.js, что удобно, поскольку Node.js уже был загружен при установке Emscripten SDK. Node.js поставляется с инструментом *npm* (Node Package Manager), который представляет собой менеджер пакетов для языка JavaScript. Доступно огромное количество пакетов (более 350 000), включая Mocha и Chai. (Чтобы получить дополнительную информацию, вы можете поискать пакеты npm на сайте www.npmjs.com.)

Чтобы установить Mocha локально в целях использования в текущем проекте, вам сначала понадобится файл `package.json`.

13.1.1. Файл package.json

Чтобы создать файл `package.json`, вы можете использовать команду `npm init`. Она выведет несколько вопросов о вашем проекте. Если для вопроса задано значение

по умолчанию, то оно будет указано в скобках. Вы можете ввести собственное значение в качестве ответа или нажать клавишу Enter, чтобы принять значение по умолчанию.

В папке WebAssembly создайте папку Chapter 13\13.2 tests\. Откройте командную строку, перейдите в папку 13.2 tests\ и запустите команду `npm init`. Укажите следующие значения:

- в качестве `package name` введите `tests`;
- в качестве `test command` введите `mocha`;
- в остальных вопросах можете принять значения по умолчанию.

Теперь в папке 13.2 tests\ появится файл `package.json`, содержимое которого показано в листинге 13.1. Свойство `test` в разделе `scripts` указывает, какой инструмент запускать при выполнении команды `npm test` в папке 13.2 tests\. В этом случае команда `test` запустит Mocha.

Листинг 13.1. Содержимое созданного файла package.json

```
{
  "name": "tests",
  "version": "1.0.0",
  "description": "",
  "main": "tests.js",
  "scripts": {
    "test": "mocha"
  },
  "author": "",
  "license": "ISC"
}
```

Mocha будет запущен, когда вы воспользуетесь командой npm test

Теперь, когда файл `package.json` создан, вы можете установить Mocha и Chai.

13.1.2. Установка Mocha и Chai

Чтобы установить Mocha и Chai для использования в текущем проекте, откройте командную строку, перейдите в папку Chapter 13\13.2 tests\, а затем выполните следующую команду, чтобы добавить их в качестве зависимостей в файл `package.json`:

```
npm install --save-dev mocha chai
```

После того как Mocha и Chai будут установлены, вы можете начать учиться писать и запускать тесты.

13.2. СОЗДАНИЕ И ЗАПУСК ТЕСТОВ

На рис. 13.1 графически представлены следующие высокоуровневые шаги, с помощью которых вы будете создавать и выполнять тесты для ваших модулей WebAssembly.

1. Напишите тесты.
2. Запустите тесты из командной строки.
3. Создайте HTML-страницу, которая загружает тесты.
4. Запустите тесты в браузерах, которые собираетесь поддерживать.

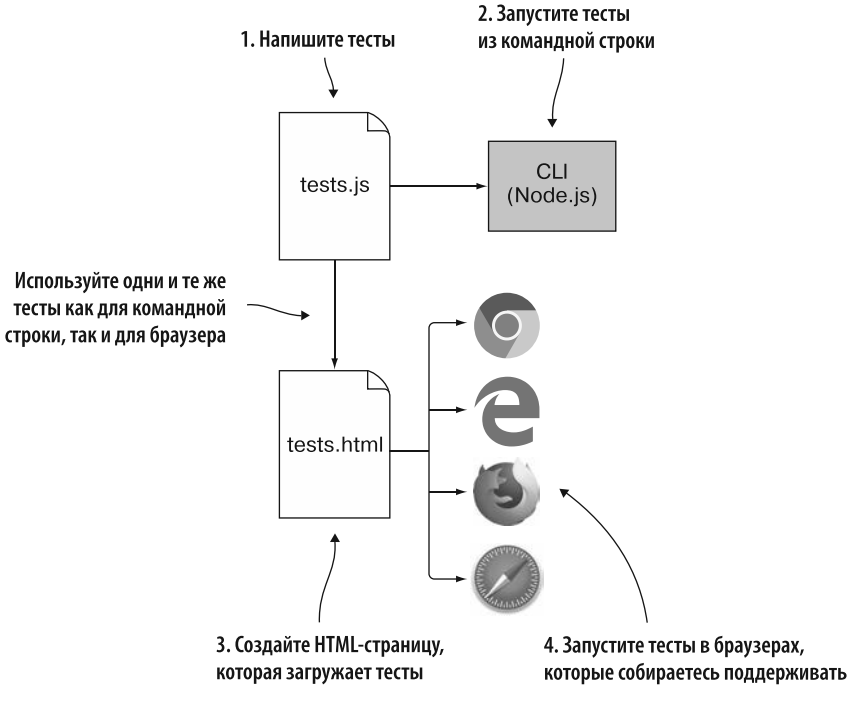


Рис. 13.1. Шаги по созданию тестов и их последующему запуску из командной строки и в браузерах, которые вы собираетесь поддерживать

13.2.1. Написание тестов

В этой главе вы напишете несколько тестов для созданного в главе 4 модуля WebAssembly, проверяющих введенные название товара и категорию. В папке 13.2 tests\:

- скопируйте файлы `validate.js` и `validate.wasm` из папки Chapter 4\4.1 js_plumbing\frontend\;
- создайте файл `tests.js`, а затем откройте его в редакторе.

Вместо того чтобы создавать два набора тестов: для командной строки и браузера, создадим один. Это сэкономит время и силы, поскольку не придется поддерживать два набора, проверяющих одно и то же.

Есть некоторые различия между запуском тестов из командной строки и в браузере, так как Mocha использует Node.js в первом случае. Начните с написания строки кода, чтобы проверить, выполняется ли тест в Node.js. Добавьте следующий фрагмент кода в файл `tests.js`:

```
const IS_NODE = (typeof process === 'object' &&
    typeof require === 'function');
```

Вашим тестам необходим доступ к библиотеке утверждений Chai, а также к объекту `Module`, созданному с помощью кода JavaScript в Emscripten. При запуске в Node.js тесты должны будут загрузить эти библиотеки, используя метод `require` в методе `before` в Mocha (метод `before` будет объяснен чуть позже). На данный момент вам нужно определить переменные, чтобы впоследствии они были доступны для вашего кода.

Добавьте код, показанный ниже, после строки `const IS_NODE` в файле `tests.js`. Совсем скоро мы добавим код к условию `else`:

```
if (IS_NODE) {    <-- Тесты запущены в Node.js
  let chai = null;
  let Module = null;
}
else {    <-- Тесты запущены в браузере
}
```

Когда вы работаете в браузере, объекты `chai` и `Module` будут созданы при добавлении этих библиотек JavaScript с помощью тега `Script` в HTML. Объект `Module` может быть не готов к взаимодействию, если вы включите файл JavaScript на веб-страницу, а затем немедленно сообщите Mocha о запуске. Чтобы гарантировать готовность данного объекта к использованию, необходимо создать объект `Module`, который JavaScript в Emscripten будет видеть при его инициализации. Внутри объекта вы определяете функцию `onRuntimeInitialized`, которая при вызове JavaScript в Emscripten сообщает платформе Mocha о необходимости запуска тестов.

Добавьте код, показанный ниже, в условие `else` оператора `if`, только что созданного в файле `tests.js`:

```
var Module = {
  onRuntimeInitialized: () => { mocha.run(); }
};
```

Когда Emscripten сигнализирует, что модуль готов к взаимодействию, запускайте тесты

Определив, где запущены тесты и объявлены необходимые глобальные переменные, можно приступить к непосредственному созданию тестов.

Функция describe

Mocha использует функцию `describe` для хранения набора тестов. Первый параметр функции — значимое имя, а второй — функция, которая выполняет один или несколько тестов.

При желании вы можете иметь вложенные функции `describe`. Например, можно использовать вложенную функцию `describe`, чтобы сгруппировать несколько тестов для одной из функций вашего модуля.

Добавьте следующую функцию `describe` в файл `tests.js` после оператора `if`:

```
describe('Testing the validate.wasm module from chapter 4', () => {
});
```

Добавив функцию `describe` для хранения коллекции тестов, вы можете изменить функцию так, чтобы при запуске тестов убедиться, что в них есть все необходимое.

Функции до и после захвата

Mocha имеет ряд специальных функций, которые могут использоваться тестами для установки предварительных условий. Например, чтобы убедиться в наличии всех нужных для запуска элементов или очистить память после запуска тестов, используются такие функции:

- `before` — запускается перед всеми тестами функции `describe`;
- `beforeEach` — выполняется перед каждым тестом;
- `afterEach` — запускается после каждого теста;
- `after` — выполняется после всех тестов в функции `describe`.

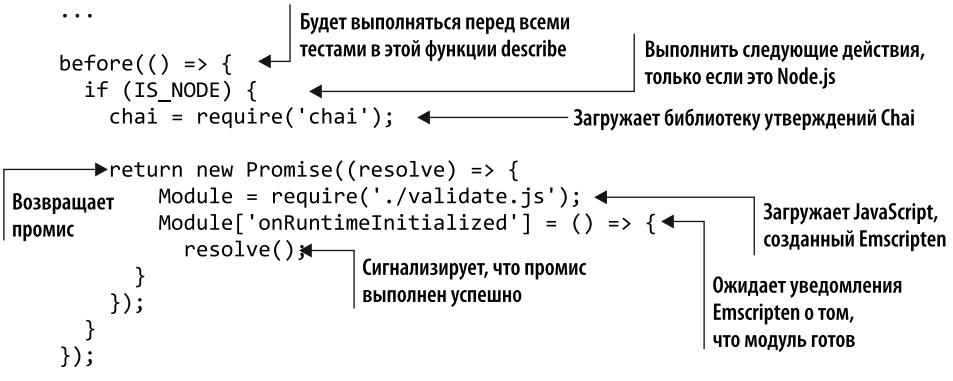
Для наших тестов необходимо реализовать функцию `before`, чтобы загрузить библиотеку Chai и модуль WebAssembly, если тесты выполняются в Node.js. Поскольку создание экземпляра модуля WebAssembly происходит асинхронно, необходимо определить функцию `onRuntimeInitialized`, чтобы получать уведомление с помощью кода JavaScript в Emscripten, когда модуль готов к взаимодействию.

СПРАВКА

Если вернуть объект Promise из функции Mocha (например, из функции before), то Mocha будет ждать завершения промиса, прежде чем продолжить.

Добавьте в файле `tests.js` код, показанный в листинге 13.2, в функцию `describe`.

Листинг 13.2. Функция before



Теперь, когда все настроено для тестирования, пора написать сам тест.

Функция it

Mocha использует функцию `it` для самих тестов. Первый параметр функции — имя теста, а второй параметр — функция, которая выполняет код для теста. Первый создаваемый нами тест подтвердит, что функция `ValidateName` в модуле возвращает правильное сообщение об ошибке, если для имени указана пустая строка. Библиотека утверждений Chai позволяет убедиться, что возвращаемое сообщение соответствует ожидаемому.

При разработке через тестирование (test-driven development, TDD) тест пишется до непосредственного написания тестируемого кода — тест не проходит, поскольку функция еще не реализована. Затем код реорганизуется таким образом, чтобы тест прошел, создается следующий тест, и процесс повторяется. «Падающие» тесты служат руководством при создании функции.

Поскольку это книга, сейчас процесс обратный — реализация выполнена до тестов. Поэтому нам нужно, чтобы тесты сначала не прошли проверку, — так мы убедимся, что они проверяют ожидаемое поведение только при успешном прохождении. После того как вы запустите тест и убедитесь, что он не прошел, можно исправить проблему так, чтобы он прошел успешно. Чтобы новый тест не прошел, вы будете использовать слово `"something"` в качестве ожидаемого сообщения об ошибке, но можно указать любую строку по своему усмотрению, если она не совпадает с возвращаемой.

Добавьте код, показанный в листинге 13.3, в функцию `describe` и после функции `before`.

Листинг 13.3. Тестирование функции ValidateName с пустой строкой в качестве имени

```
...

it("Pass an empty string", () => {
  const errorMessagePointer = Module._malloc(256);
  const name = "";
  const expectedMessage = "something";

  const isValid = Module.ccall('ValidateName',
      'number',
      ['string', 'number', 'number'],
      [name, 50, errorMessagePointer]);

  let errorMessage = "";
  if (isValid === 0) {
    errorMessage = Module.UTF8ToString(errorMessagePointer);
  }

  Module._free(errorMessagePointer);

  chai.expect(errorMessage).to.equal(expectedMessage);
});
```

Определяет сам тест

Устанавливает имя как пустую строку

Ожидаемое сообщение об ошибке; намеренно сделано неправильным, поэтому тест не проходит

Вызывает функцию ValidateName в модуле

Если произошла ошибка, то считывает сообщение об ошибке из памяти модуля

Проверяет, соответствует ли возвращенное сообщение ожидаемому

Второй тест, который мы создадим, подтвердит, что функция `ValidateName` возвращает правильное сообщение об ошибке, если имя слишком длинное. Чтобы добавить этот тест, сделайте следующее:

- сделайте копию своего первого теста и вставьте эту копию под первым;
- измените имя в функции `it` на `"Pass a string that's too long"`;
- установите для переменной имени значение `"Longer than 5 characters"`;
- измените значение, переданное для второго параметра функции `ValidateName`, с `50` на `5`.

Ваш новый тест теперь должен соответствовать коду, показанному в листинге 13.4.

Листинг 13.4. Тестирование функции ValidateName со слишком длинным именем

```
...

it("Pass a string that's too long", () => {
  const errorMessagePointer = Module._malloc(256);
  const name = "Longer than 5 characters";
  const expectedMessage = "something";
```

Измените название теста, чтобы оно отражало то, что вы тестируете

Предоставляет имя длиной более пяти символов

```
  const isValid = Module.ccall('ValidateName',
      'number',
      ['string', 'number', 'number'],
      [name, 5, errorMessagePointer]);

  let errorMessage = "";
  if (isValid === 0) {
    errorMessage = Module.UTF8ToString(errorMessagePointer);
  }

  Module._free(errorMessagePointer);

  chai.expect(errorMessage).to.equal(expectedMessage);
});
```

Сообщает функции, что максимальная длина строки составляет пять символов

Поздравляю! Вы написали свой первый набор тестов WebAssembly. Следующий шаг — запустить их.

13.2.2. Запуск тестов из командной строки

Следующий шаг — запустить тесты из командной строки. Для этого откройте командную строку, перейдите в папку Chapter 13\13.2 tests\ и выполните такую команду:

```
npm test tests.js
```

На рис. 13.2 показаны результаты тестов: тесты отмечены номером, если не прошли, и галочкой — если были успешны. Упавшие тесты также перечислены в итоговом разделе, в котором подробно рассказывается, почему именно не прошли. В этом случае все тесты не прошли, поскольку мы намеренно указали неверные значения для ожидаемого результата.

Прежде чем исправлять тесты так, чтобы они прошли, создадим HTML-страницу, чтобы их можно было запускать и в браузере.

13.2.3. HTML-страница, загружающая тесты

Как показано на рис. 13.3, в этом подразделе мы создадим HTML-страницу, которая позволит запускать тесты в браузере. Будем использовать в браузере те же тесты, что и в командной строке. Возможность применять одни и те же тесты и там и там экономит усилия, поскольку не приходится поддерживать два набора тестов для одного и того же функционала.

В папке 13.2 tests\ создайте файл `tests.html` и откройте его в редакторе.

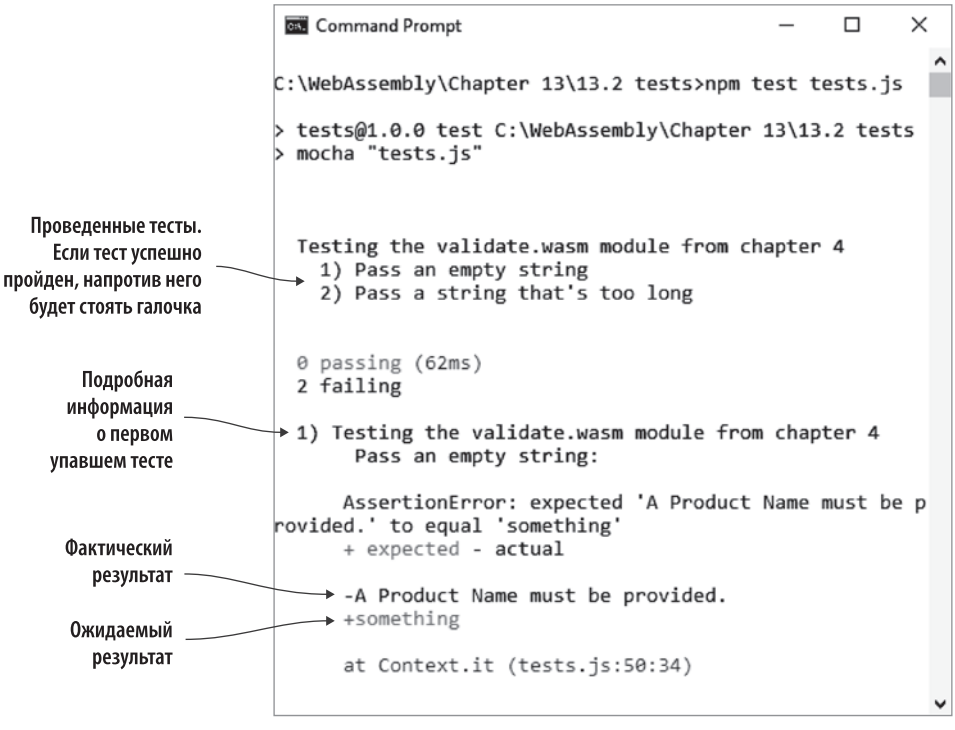


Рис. 13.2. Результаты тестов в командной строке. Оба теста не прошли, поскольку вы намеренно предоставили неверную ожидаемую строку 'something'

Рис. 13.3. Следующий шаг — создание HTML-страницы, чтобы тесты можно было запускать и в браузере

СПРАВКА

HTML-файл, который мы собираемся создать, был скопирован с сайта Mocha и немного изменен. Исходный файл можно найти по адресу https://mochajs.org/#running-mocha-in-the-browser.

При запуске в браузере библиотека утверждений Chai и модуль WebAssembly загружаются путем добавления их в теги `Script`. При запуске в Node.js они загружаются с помощью метода `require`. Области, которые были изменены в шаблоне HTML Mocha, находятся после тега `Script` с классом `"mocha-init"`. Теги `Script` для `test.array.js`, `test.object.js` и `test.xhr.js`, а также класс `"mocha-exec"` были заменены тегом `Script` для тестового файла `tests.js` и созданного Emscripten файла JavaScript `validate.js`.

Обратите внимание: файл `tests.js` необходимо включить в HTML перед файлом JavaScript, созданным Emscripten (`validate.js`). Это связано с тем, что код включен в файл `tests.js` с целью сообщить Emscripten о необходимости вызова функции `onRuntimeInitialized` по мере готовности модуля. Когда эта функция будет вызвана, код станет запускать Mocha-тесты.

Добавьте код, показанный в листинге 13.5, в файл `tests.html`.

Листинг 13.5. HTML-код файла tests.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Mocha Tests</title>
    <meta name="viewport"
        content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="https://unpkg.com/mocha/mocha.css" />
  </head>
  <body>
    <div id="mocha"></div>

    <script src="https://unpkg.com/chai/chai.js"></script>
    <script src="https://unpkg.com/mocha/mocha.js"></script>

    <script class="mocha-init">
      mocha.setup('bdd');
      mocha.checkLeaks();
    </script>

    <script src="tests.js"></script>
    <script src="validate.js"></script>
  </body>
</html>
```

Ваши тесты (должны быть включены до созданного Emscripten файла JavaScript)

Файл JavaScript, созданный Emscripten

После того как HTML-файл будет создан, можно перейти к запуску тестов в браузере.

13.2.4. Запуск тестов из браузера

Как показано на рис. 13.4, теперь мы запустим те же тесты, что и в командной строке, однако на этот раз в браузере. Откройте браузер и введите `http://localhost:8080/tests.html` в адресную строку, чтобы увидеть результаты тестов, как показано на рис. 13.5.

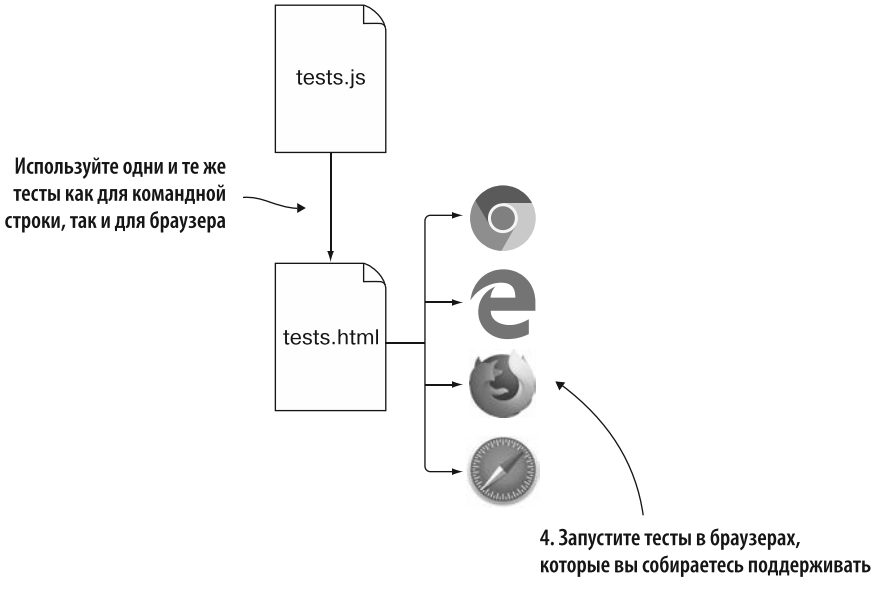


Рис. 13.4. Следующий шаг — запустить тесты в браузере

Теперь, когда ваши тесты выполняются в командной строке и в браузерах, вы можете настроить их так, чтобы они проходили успешно.

13.2.5. Исправление тестов

Убедившись в выполнении тестов, вы можете изменить их так, чтобы они проходили успешно. Откройте файл `tests.js` и внесите следующие изменения:

- в тесте `"Pass an empty string"` установите для параметра `expectedMessage` значение `"A Product Name must be provided"`;
- в тесте `"Pass a string that's too long"` установите в качестве значения ожидаемого сообщения пустую строку (`""`) и измените значение, переданное в качестве второго параметра функции модуля `ValidateName`, с `5` на `50`.

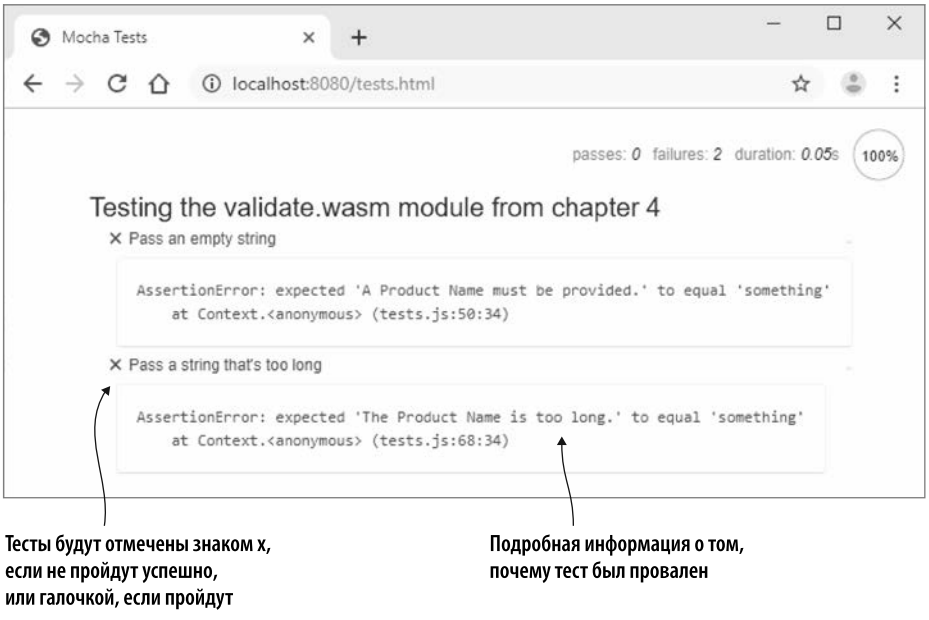


Рис. 13.5. Результаты тестов, запущенных в браузере

Теперь нужно убедиться, что тесты проходят успешно. В командной строке перейдите в папку Chapter 13\13.2 tests\, а затем выполните следующую команду:

```
npm test tests.js
```

Тесты должны пройти, как показано на рис. 13.6.

Рис. 13.6. Оба теста теперь проходят успешно при запуске из командной строки

Вы также можете убедиться, что тесты проходят в браузере, введя `http://localhost:8080/tests.html` в адресную строку, чтобы увидеть результаты, как показано на рис. 13.7.

Рис. 13.7. Результаты тестов, запущенных в браузере, показывают, что тесты прошли успешно

13.3. ЧТО ДАЛЬШЕ?

WebAssembly не сидел на месте с тех пор, как в 2017 году получил статус MVP. После MVP введение функции `WebAssembly.instantiateStreaming` обеспечило более быструю компиляцию и создание экземпляров, была добавлена возможность импорта или экспорта изменяемых глобальных объектов, в настольной версии браузера Chrome появилась поддержка pthread, и улучшения в браузерах все еще продолжаются.

Группа сообщества WebAssembly усердно работает над функциями, которые будут добавлены в WebAssembly, чтобы другие языки программирования могли использовать его более легко и открывать еще больше сценариев использования. Список предложений касательно функций WebAssembly и их текущий статус можно найти на странице https://github.com/WebAssembly/proposals.

Кроме того, началась работа над спецификацией WASI, призванная стандартизировать то, как WebAssembly будет работать вне браузера. В Mozilla есть хорошая статья Лин Кларк о WASI: *Standardizing WASI: A system interface to run WebAssembly outside the web* (http://mng.bz/O9Pa).

Поскольку WebAssembly будет продолжать улучшаться и расширяться, ниже представлены несколько вариантов, которыми вы можете воспользоваться, чтобы найти помощь в случае возникновения проблемы.

- Документация Emscripten находится на сайте https://emscripten.org.
- Если вы обнаружите проблему с самим Emscripten, то можете проверить, не отправил ли кто-то отчет об ошибке или не знает ли кто-то, как ее решить, на странице https://github.com/emscripten-core/emscripten.
- Emscripten имеет очень активное сообщество с частыми выпусками обновлений. Если доступна более новая версия Emscripten, то можете попробовать обновиться до последней версии, чтобы увидеть, решит ли это вашу проблему. Инструкции по обновлению находятся в приложении А.
- В Mozilla Developer Network есть хорошая документация по WebAssembly: https://developer.mozilla.org/en-US/docs/WebAssembly.
- Не стесняйтесь оставлять комментарии в liveBook к этой книге по адресу https://livebook.manning.com/#!Book/webassembly-in-action/welcome.
- Следите за мной в Twitter (@Gerard_Gallant) и в моем блоге, пока я продолжаю изучать все, что может предложить WebAssembly: https://cggallant.blogspot.com.

УПРАЖНЕНИЯ

Решения этих упражнений можно найти в приложении Г.

1. Какую функцию Mocha вы бы использовали, если бы хотели сгруппировать несколько связанных тестов?
2. Напишите тест, чтобы убедиться в том, что при передаче пустой строки для значения `categoryId` в функции `ValidateCategory` возвращается правильное сообщение об ошибке.

РЕЗЮМЕ

В этой главе вы узнали следующее.

- Написание автоматических тестов занимает немного времени, но, будучи написанными, они могут работать быстро, выполняться так часто, как вам нужно, запускаться в любое время, работать каждый раз одинаково и выполняться на разных платформах.
- Автоматические тесты не устраняют необходимость ручного тестирования, но могут обрабатывать однообразные элементы, позволяя вам сосредоточиться на других областях.
- Mocha — один из нескольких доступных фреймворков тестирования JavaScript, поддерживающий любую библиотеку утверждений. Вдобавок он может за-

пускать тесты как из командной строки, так и в браузере. При запуске из командной строки Mocha использует Node.js для запуска тестов.

- В Mocha тесты группируются с помощью функции `describe`, а сами тесты используют функцию `it`.
- У Mocha есть несколько специальных функций-обработчиков (`before`, `beforeEach`, `afterEach` и `after`), которые вы можете использовать для установки предварительных условий перед запуском тестов и для очистки после тестов.
- Когда промис возвращается из функций Mocha, Mocha ожидает выполнения промиса, прежде чем продолжить. Это полезно, если у вас есть асинхронные операции.
- Если тест не прошел, то у вас есть детальная информация о том, почему он не прошел.
- Если тест пройден, то в журнале рядом с ним отображается галочка.

Приложения

A Установка и настройка инструментов

В этом приложении

- ✓ Установка Python.
- ✓ Запуск локального веб-сервера с использованием Python.
- ✓ Проверка того, настроен ли тип носителя WebAssembly для Python, и если нет, то изучение его настройки.
- ✓ Скачивание и установка Emscripten SDK.
- ✓ Обзор набора инструментов WebAssembly Binary Toolkit.

В этом приложении вы установите и настроите все инструменты, необходимые для работы с примерами, приведенными в данной книге. Основной инструмент, который вам понадобится, — Emscripten. Первоначально созданный для преобразования кода на C и C++ в asm.js, он был модифицирован для компиляции кода в модули WebAssembly.

A.1. PYTHON

Запуск установки Emscripten SDK в системе требует установки Python. Базовая необходимая версия Python — 2.7.12. Вы можете проверить, установлен

ли уже Python и какая у него версия, выполнив следующую команду в окне консоли:

```
python -V
```

Если Python установлен, то вы должны увидеть сообщение, похожее на то, которое показано на рис. A.1.

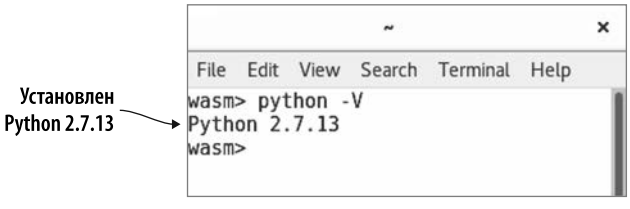


Рис. A.1. Проверка установки Python

Если Python не установлен, то вы можете загрузить пакет для его установки со страницы www.python.org/downloads/. Если вы используете версию Linux с APT (расширенный пакет Tool), то Python также можно установить, выполнив следующую команду в окне терминала:

```
sudo apt install python-minimal
```

A.1.1. Запуск локального веб-сервера

Большинство примеров в этой книге потребуют от вас использования локального *веб-сервера*, поскольку некоторые браузеры по умолчанию не разрешают доступ к файловой системе для загрузки других файлов. Это помешает работе отдельных функций JavaScript API в WebAssembly в определенных браузерах, если файл HTML запускается непосредственно из файловой системы.

ОПРЕДЕЛЕНИЕ

Веб-сервер — это специальная программа, которая с помощью HTTP передает файлы, используемые веб-страницами, вызывающей стороне (в нашем случае браузеру).

Удобно, что Python может запускать локальный веб-сервер и что есть два способа запустить его в зависимости от установленной версии Python. Для обоих подходов вы открываете окно консоли, переходите к папке, в которой находится файл HTML, а затем запускаете команду.

Если вы используете Python 2.x, то следующая команда запускает локальный веб-сервер:

```
python -m SimpleHTTPServer 8080
```

Команда для Python 3.x:

```
python3 -m http.server 8080
```

Вы увидите сообщение о том, что HTTP обслуживается через порт 8080, как показано на рис. А.2.

Рис. А.2. Локальный веб-сервер Python 2.x, работающий на порте 8080

Теперь все, что вам нужно сделать, — открыть браузер и перейти по адресу http://localhost:8080/, за которым следует имя файла HTML, который нужно просмотреть.

Другой доступный вариант — задействовать инструмент emrun, который поставляется с Emscripten. Emrun запускает локальный веб-сервер Python, а затем запускает файл, указанный в вашем браузере по умолчанию. Ниже приведен пример использования команды emrun для запуска файла `test.html`:

```
emrun --port 8080 test.html
```

ПРИМЕЧАНИЕ

Для всех трех команд путь, по которому обслуживаются файлы, будет зависеть от каталога, в котором вы находитесь при запуске локального веб-сервера.

А.1.2. Тип носителя WebAssembly

Тип носителя (media type) изначально был известен как тип MIME. MIME означает Multipurpose Internet Mail Extensions (многоцелевые расширения электронной почты в Интернете) и использовался для обозначения типа содержания сообщений электронной почты. Браузеры также используют тип носителя файла, чтобы определить, как обрабатывать файл.

Первоначально файлы WebAssembly передавались в браузеры с помощью типа носителя `application/octet-stream`, поскольку файл `.wasm` представляет собой двоичные данные. С тех пор он был изменен на более формальный тип носителя: `application/wasm`.

К сожалению, регистрация новых типов носителей в IANA (Internet Assigned Numbers Authority, полномочный орган по цифровым адресам в Интернет), который отвечает за стандартизацию типов носителей, требует времени. Из-за этого не все веб-серверы включают тип носителя WebAssembly, и потому вам необходимо убедиться, что он определен для вашего веб-сервера, чтобы браузер знал, что делать с модулями WebAssembly.

Не обязательно использовать в качестве локального веб-сервера именно Python, если вы предпочитаете какой-то другой веб-сервер. Поскольку Python был установлен для Emscripten SDK, это удобно, если на вашем компьютере не установлены другие веб-серверы. На Mac или Linux, прежде чем пытаться добавить тип носителя WebAssembly в список типов носителя Python, вы можете проверить и увидеть, существует ли он уже, выполнив следующую команду:

```
grep 'wasm' /etc/mime.types
```

Если расширение wasm еще не добавлено в Python, то ничего отображаться не будет. В противном случае вы должны увидеть нечто похожее на снимок экрана, представленный на рис. A.3.

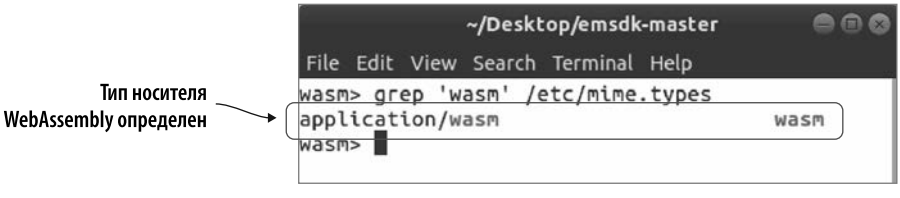


Рис. A.3. Тип носителя WebAssembly — часть списка типов носителей Python в Ubuntu Linux

На Mac или Linux, если тип мультимедиа еще не добавлен в Python, вы можете добавить его вручную, отредактировав файл `mime.types`. Следующая команда использует gedit в качестве редактора, но если он недоступен, можно заменить gedit в этой команде каким-либо из других редакторов:

```
sudo gedit /etc/mime.types
```

Добавьте следующий элемент в список типов носителя, а затем сохраните и закройте файл:

```
application/wasm   wasm
```

Под Windows, чтобы проверить, настроен ли в Python тип носителя, вам нужно проверить файл `mimetypes.py`. Если вы откроете окно консоли и перейдете в папку

Lib, в которой установлен Python, то сможете проверить, присутствует ли в файле тип носителя WebAssembly, с помощью такой команды:

```
type mimetypes.py | find "wasm"
```

Если расширение wasm еще не добавлено в Python, то ничего отображаться не будет. В противном случае вы должны увидеть нечто похожее на рис. А.4.

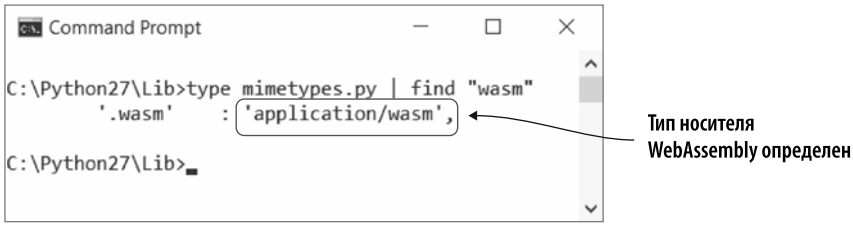


Рис. А.4. Тип носителя WebAssembly также есть в списке типов носителя Python в Windows

Если тип носителя не указан в этом файле, то необходимо отредактировать файл. Откройте его в любом редакторе. Поиск по тексту `types_map = {` должен привести вас к разделу файла, в котором нужно добавить тип носителя, как показано на рис. А.5.

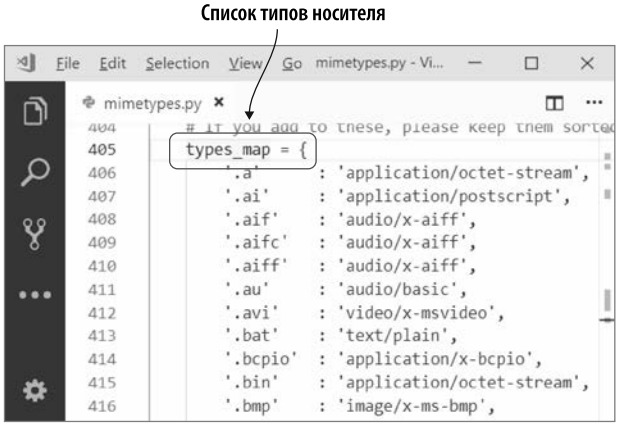


Рис. А.5. Раздел types_map в файле mimetypes.py, открытый с помощью Visual Studio Code

Добавьте следующий элемент в список в разделе `types_map`, а затем сохраните и закройте файл:

```
'.wasm'   : 'application/wasm',
```

A.2. EMSCRIPTEN

На момент написания данной книги последней версией Emscripten SDK являлась 1.38.45. Набор инструментов регулярно обновляется, поэтому у вас может быть более новая версия.

Прежде чем приступить к загрузке и установке SDK, вы должны проверить, не установлен ли он уже. Для этого запустите следующую команду в окне консоли, чтобы просмотреть список инструментов, которые были установлены с SDK:

```
emsdk list
```

Если SDK установлен, то вы должны увидеть список, аналогичный показанному на рис. A.6. Если SDK установлен и имеет версию, необходимую для этой книги (или выше), то вы можете перейти к разделу A.3.

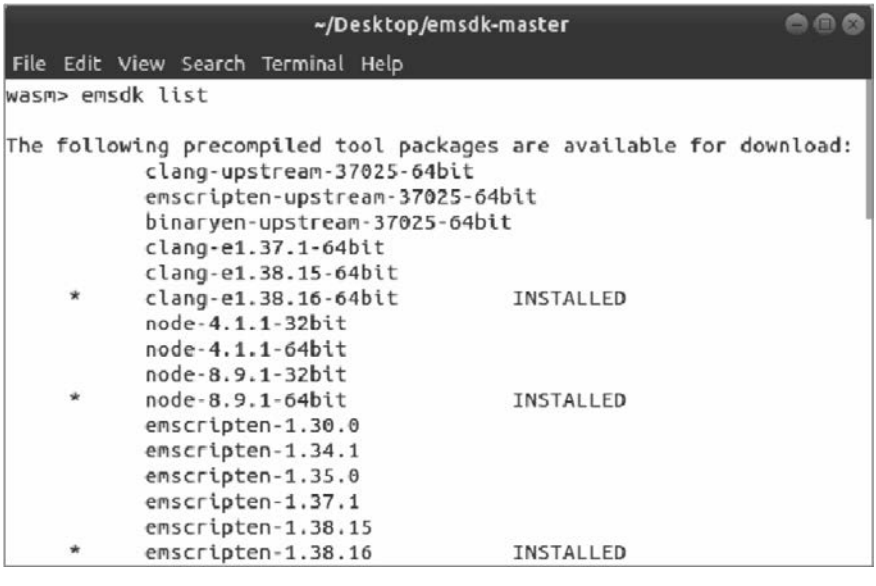


Рис. A.6. Установлен Emscripten SDK версии 1.38.16

Если SDK установлен, но не той версии, которая вам нужна для этой книги, то выполните следующую команду, чтобы дать SDK указание получить новейший список доступных инструментов:

```
emsdk update
```

Вы можете пропустить следующий подраздел и перейти к подразделу A.2.2, если используете Windows, или подразделу A.2.3, если используете Mac или Linux.

Если SDK не установлен, то нужно сделать следующий шаг — загрузить Emscripten SDK.

А.2.1. Загрузка Emscripten SDK

Перейдите на сайт https://github.com/emscripten-core/emsdk. Нажмите зеленую кнопку Clone or Download (Клонировать или скачать), расположенную в правой части экрана[1], а затем щелкните на ссылке Download ZIP (Скачать ZIP) во всплывающем окне, как показано на рис. А.7.

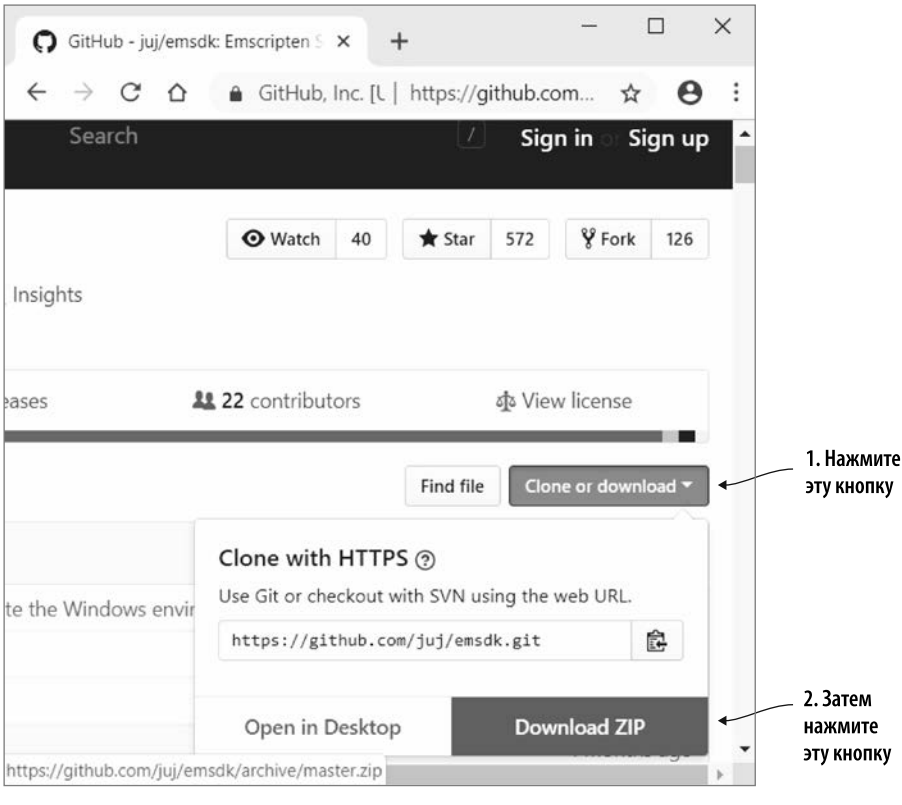


Рис. А.7. Нажмите кнопку Clone or Download (Клонировать или скачать), а затем нажмите кнопку Download ZIP (Скачать ZIP), чтобы скачать Emscripten SDK

[1] Этот способ все еще работает, но рекомендуемый способ установки/обновления уже другой — через git clone/git pull. — *Примеч. науч. ред.*

Распакуйте файлы в желаемое место. Затем откройте окно консоли и перейдите к извлеченной папке emsdk-master.

A.2.2. Если вы используете Windows

Следующая команда загрузит последние инструменты SDK:

```
emsdk install latest
```

Выполните команду, приведенную ниже, чтобы сделать последний пакет SDK активным для текущего пользователя. Вам может потребоваться открыть окно консоли от имени администратора, поскольку ей потребуется доступ к реестру Windows в случае применения параметра `--global`:

```
emsdk activate latest --global
```

СПРАВКА

Параметр --global необязателен, но рекомендуется, чтобы переменные среды также помещались в реестр Windows. Если параметр не используется, то файл emsdk_env.bat необходимо будет запускать каждый раз при открытии нового окна консоли, чтобы инициализировать переменные среды.

A.2.3. Если вы используете Mac или Linux

Выполните эту команду, чтобы загрузить последние инструменты SDK:

```
./emsdk install latest
```

Выполните следующую команду, чтобы активировать последнюю версию SDK:

```
./emsdk activate latest
```

Вам нужно будет выполнить такую команду, чтобы текущее окно терминала знало переменные среды:

```
source ./emsdk_env.sh
```

Самое большое преимущество этой команды — вам больше не нужно добавлять к командам префиксы, такие как `emsdk`, с помощью символов `./`. К сожалению, переменные среды не кэшируются, поэтому придется запускать команду каждый раз, когда вы открываете новое окно терминала. Либо вы можете поместить команду в свой `.bash_profile` или аналогичный файл. При добавлении команды в `.bash_profile` или аналогичный файл вам нужно будет изменить путь в зависимости от того, где была размещена папка emsdk-master.

А.2.4. Решение проблем при установке

Если у вас возникнут проблемы с установкой, то на сайте https://emscripten.org/docs/getting_started/downloads.html есть инструкции по установке Emscripten на Windows, Mac и Linux, которые могут помочь.

Иногда загрузка и установка Emscripten SDK может не работать из-за конфликтов с существующими системными библиотеками на вашем компьютере. В этом случае вам может потребоваться собрать Emscripten из исходников. Вы можете найти соответствующие инструкции на https://emscripten.org/docs/building_from_source/index.html.

А.3. NODE.JS

Emscripten SDK при установке загружает несколько инструментов в дополнение к Emscripten, один из которых — Node.js. Это среда выполнения JavaScript, построенная на движке V8, на котором также работает браузер Chrome. Node.js позволяет использовать JavaScript в качестве кода на стороне сервера, а также имеет большое количество пакетов с открытым исходным кодом, доступных для решения многих задач программирования. Модули WebAssembly можно использовать в Node.js, поэтому мы включим в данную книгу несколько примеров для Node.js.

Поддержка WebAssembly была добавлена в Node.js в версии 8, так что это базовая необходимая версия. Выполните следующую команду, чтобы просмотреть список инструментов, которые были установлены при загрузке Emscripten SDK.

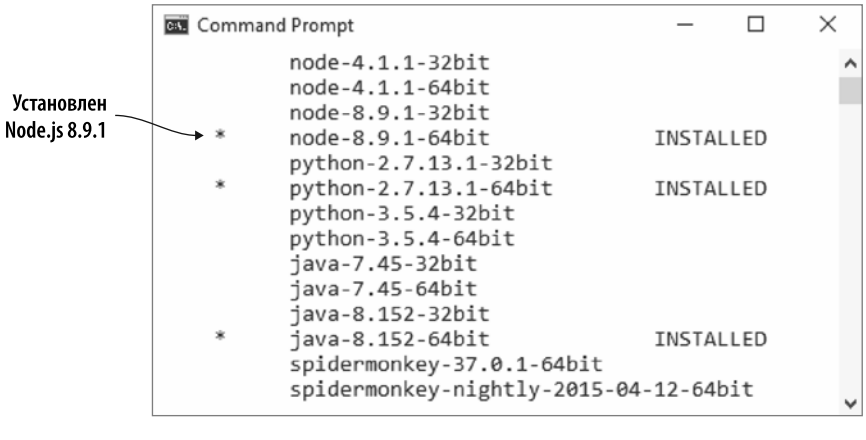


Рис. А.8. Node.js версии 8.9.1 был установлен вместе с Emscripten SDK

Вы должны увидеть нечто похожее на рис. A.8, на котором указана установленная версия Node.js:

```
emsdk list
```

Если версия Node.js, установленная с SDK, не является версией 8 или выше, то необходимо удалить ее из SDK. Для этого в командной строке введите `emsdk uninstall` и полное имя установленной версии Node.js:

```
emsdk uninstall node-4.1.1-64bit
```

После того как Node.js 4 будет удален, можно использовать команду установки `emsdk`, чтобы установить Node.js версии 8.9, которая была указана как доступная для загрузки при запуске списка `emsdk`:

```
emsdk install node-8.9.1-64bit
```

A.4. WEBASSEMBLY BINARY TOOLKIT

WebAssembly Binary Toolkit содержит инструменты, которые позволят выполнять преобразование между двоичным форматом WebAssembly и текстовым форматом. Инструмент wasm2wat преобразует двоичный формат в текстовый, а инструмент wat2wasm — наоборот. Существует даже инструмент wasm-interp, который позволяет двоичному файлу WebAssembly автономно запускаться вне браузера, что может быть полезно для автоматического тестирования модуля WebAssembly.

Браузеры будут использовать текстовый формат WebAssembly в случае просмотра исходного кода или отладки, если в модуль WebAssembly не включены карты исходников, поэтому важно иметь базовое понимание текстового формата. Так, вы будете работать с текстовым форматом, чтобы создать игру в главе 11.

Карты исходников — это файлы, которые отображают текущий код, который мог быть изменен и переименован в процессе компиляции, в исходный код, чтобы отладчики могли восстановить отлаживаемый код до чего-то более близкого к оригиналу, что облегчает отладку.

Скачать выполняемые файлы WebAssembly Binary Toolkit не получится. Чтобы получить копию, нужно клонировать репозиторий, который находится на GitHub, а затем создать выполняемые файлы напрямую. Если вам неудобно использовать git, то в репозитории GitHub этого инструментария есть несколько демоверсий, с которыми вы можете работать с помощью своего браузера:

- демоверсия wat2wasm позволяет ввести текстовый формат и загрузить файл Wasm: https://webassembly.github.io/wabt/demo/wat2wasm;

- демоверсия wasm2wat позволяет загрузить файл Wasm и просмотреть текстовый формат: https://webassembly.github.io/wabt/demo/wasm2wat.

Работать с примерами в этой книге можно с помощью онлайн-демоверсии wat2wasm, но при желании можно скачать исходный код для набора инструментов и собрать файлы Wasm локально. Инструкции по клонированию и сборке набора инструментов доступны по адресу https://github.com/WebAssembly/wabt.

A.5. BOOTSTRAP

Чтобы веб-страница выглядела более профессионально, вместо того чтобы стилизовать все вручную, будем использовать Bootstrap. Это популярный фреймворк для веб-разработки, который включает ряд шаблонов дизайна, помогающих упростить и ускорить веб-разработку. Примеры в этой книге будут просто указывать на файлы, которые размещены на CDN, но Bootstrap можно скачать с сайта https://getbootstrap.com, если вы предпочитаете локальную копию.

СПРАВКА

Сеть доставки контента (content delivery network, CDN) географически распределена, чтобы обслуживать необходимые файлы как можно ближе к устройству, запрашивающему их. Это распределение ускоряет процесс скачивания файлов, что сокращает время загрузки сайта.

Bootstrap зависит от библиотек jQuery и Popper.js. jQuery — это библиотека JavaScript, которая упрощает работу с DOM, событиями, анимацией и Ajax. Popper.js — это механизм позиционирования, помогающий размещать элементы на веб-странице.

Popper.js включен в файлы `bootstrap.bundle.js` и `bootstrap.bundle.min.js`, а jQuery — нет. Вам также придется скачать jQuery, если не хотите использовать CDN. Можно сделать это, перейдя по следующему адресу: https://jquery.com/download.

Б Функции ccall, cwrap и вызовы функций напрямую

В этом приложении

- ✓ Вызов функции модуля из JavaScript с помощью вспомогательных функций Emscripten ccall и cwrap.
- ✓ Вызов функции модуля непосредственно из JavaScript без применения вспомогательных функций Emscripten.
- ✓ Передача массивов в функцию.

При работе с созданным Emscripten связующим кодом JavaScript вы получаете несколько вариантов вызова модуля. Наиболее распространенный подход — использовать функции `ccall` и `cwrap`, которые помогают управлять памятью, например, при передаче и возврате строк. Вы также можете вызвать функцию модуля напрямую.

Б.1. ФУНКЦИЯ CCALL

Функция `ccall` позволяет вызывать функцию в модуле WebAssembly и получать результаты. Она принимает четыре параметра.

- Строка, указывающая *имя вызываемой функции* в модуле. Создавая модуль WebAssembly, Emscripten добавляет символ подчеркивания перед именем функции. Не включайте начальный символ подчеркивания, так как функция `ccall` будет добавлять его самостоятельно.
- *Тип возвращаемого значения* функции. Могут быть указаны следующие значения:
  - `null`, если функция возвращает `void`;
  - `'number'`, если функция возвращает `integer`, `float` или указатель;
  - `'string'`, если функция возвращает `char*`. Это не обязательно и сделано для удобства. При указании этого значения функция `ccall` будет ответственна за управление памятью возвращаемой строки.
- Массив, указывающий *типы данных параметров*. Он должен иметь такое же количество элементов, как и параметры функции, и они должны быть в том же порядке. Можно указать следующие значения:
  - `'number'`, если параметр является `integer`, `float` или указателем;
  - `'string'` может использоваться для параметра `char*`. При указании этого значения функция `ccall` будет ответственна за управление памятью строки. Значение при таком подходе считается временным, поскольку память будет освобождена в момент возврата из функции;
  - `'array'` можно использовать, но только для массивов 8-битных значений.
- Массив *значений, передаваемых функции*. Каждый его элемент соответствует параметрам функции, и элементы должны быть расположены в том же порядке.

Типы данных `string` и `array` третьего параметра используются для удобства, выполняя работу по созданию указателя, копируя значение в память, а затем освобождая эту память, после того как вызов функции завершится. Эти значения считаются временными и будут представлены только во время выполнения функции. Если код модуля WebAssembly сохраняет указатель для будущего использования, то он может указывать на недопустимые данные.

Если вы хотите, чтобы объекты жили дольше, необходимо выделить и освободить память вручную с помощью функций `_malloc` и `_free` в Emscripten. В этом случае вы не будете использовать `string` или `array` в качестве типа параметра, а задействуете `number`, поскольку будете передавать указатель напрямую, не прибегая к Emscripten в управлении памятью.

Если вам нужно передать массив, который имеет значения больше 8-битных, например 32-битные целые числа, то передайте указатель, а не тип массива. В разделе Б.3 показано, как передать массив в модуль вручную.

Б.1.1. Создание простого модуля WebAssembly

Чтобы продемонстрировать работу функции `ccall`, нам понадобится модуль WebAssembly. Создайте папку `Appendix B\B.1 ccall\` для новых файлов. Создайте в папке файл `add.c`, а затем откройте его в своем любимом редакторе. Следующий код на C для функции `Add` примет два значения, сложит их друг с другом и затем вернет результат. Поместите этот фрагмент кода в файл `add.c`:

```
#include <stdlib.h>
#include <emscripten.h>

EMSCRIPTEN_KEEPALIVE
int Add(int value1, int value2) {
  return (value1 + value2);
}
```

Вы будете повторно использовать этот модуль в следующих разделах для `cwrap` и для прямого вызова. Поскольку вам понадобятся функции `ccall` и `cwrap`, доступные в объекте `Module` в JavaScript, сгенерированном Emscripten, то необходимо добавить их как часть массива `EXTRA_EXPORTED_RUNTIME_METHODS` командной строки. Чтобы скомпилировать код в модуль WebAssembly, откройте командную строку, перейдите в папку, в которой сохранен файл `add.c`, а затем выполните следующую команду:

```
emcc add.c -o js_plumbing.js
➥ -s EXTRA_EXPORTED_RUNTIME_METHODS=['ccall','cwrap']
```

Б.1.2. Создание веб-страницы, которая будет взаимодействовать с модулем WebAssembly

Теперь вам нужно создать простую веб-страницу в формате HTML, а также включить JavaScript для вызова функции `Add` на веб-странице, а не в отдельном файле. В папке `B.1 ccall` создайте файл `add.html`, а затем откройте его в редакторе. На этой веб-странице мы разместим простую кнопку, которая при нажатии будет вызывать еще не созданную функцию JavaScript `callAdd`. Функция JavaScript вызовет функцию `Add` в модуле с помощью вспомогательной функции `ccall` в Emscripten, а затем отобразит результат сложения в окне консоли инструментов разработчика браузера. Добавьте код, показанный в листинге Б.1, в файл `add.html`.

Получив готовый код на JavaScript, вы можете открыть браузер и ввести `http://localhost:8080/add.html` в адресную строку, чтобы увидеть только что созданную веб-страницу. Откройте инструменты разработчика браузера (нажмите `F12`), чтобы просмотреть консоль, а затем нажмите кнопку `Add` (Добавить), чтобы увидеть результат вызова функции добавления модуля, как показано на рис. Б.1.

Листинг Б.1. HTML-код файла add.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <input type="button" value="Add" onclick="callAdd()" />

    <script>
      function callAdd() {
        const result = Module.ccall('Add',
            'number',
            ['number', 'number'],
            [1, 2]);

        console.log(`Result: ${result}`);
      }
    </script>

    <script src="js_plumbing.js"></script>
  </body>
</html>
```

Первый параметр — это имя функции

Передает значения параметров

Тип возвращаемого значения — целое число в модуле

Типы параметров в модуле — целые числа

Отображает результат

Файл JavaScript, созданный Emscripten

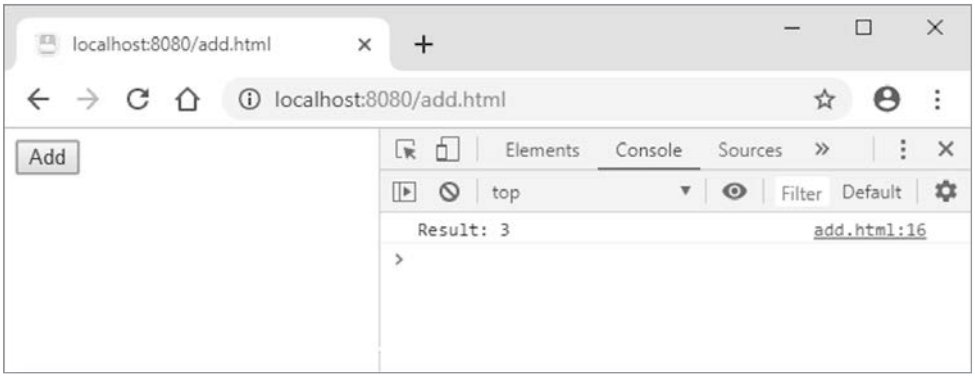


Рис. Б.1. Результат вызова функции Add модуля с помощью ccall и передачи значений параметров 1 и 2

Б.2. ФУНКЦИЯ CWRAP

Функция `cwrap` похожа на функцию `ccall`. В функции `cwrap` вы указываете только первые три параметра, которые идентичны параметрам `ccall`:

- имя функции;
- тип возвращаемого значения функции;
- массив, указывающий типы параметров функции.

В отличие от `ccall`, которая выполняет функцию сразу, при вызове функции `cwrap` вам предоставляется JavaScript-функция. В JavaScript функции являются полноправными и могут передаваться так же, как и переменные, что является одной из самых мощных особенностей JavaScript. Затем с помощью функции JavaScript можно вызвать функцию модуля, аналогично тому как вы вызываете обычную функцию, в которой указываются значения параметров напрямую, а не с помощью массива.

Б.2.1. Изменение кода JavaScript для использования cwrap

Чтобы продемонстрировать применение функции `cwrap`, создайте папку Appendix B\B.2 cwrap\ для новых файлов. Скопируйте файлы `add.html`, `js_plumbing.js` и `js_plumbing.wasm` из Appendix B\B.1 ccall\ в Appendix B\B.2 cwrap\. Откройте файл `add.html` в вашем любимом редакторе, чтобы изменить функцию `callAdd` для использования вспомогательной функции `cwrap` в Emscripten.

Поскольку `cwrap` будет возвращать функцию, а не результат функции `Add` модуля, то первое изменение, которое вам нужно сделать, — изменить переменную `const result` на `const add`. Кроме того, измените `Module.ccall` на `Module.cwrap`. Наконец, удалите четвертый параметр, в котором вы указали значения для параметров, так как функция `cwrap` принимает только первые три из них.

Определив функцию, которая может вызывать функцию `Add` модуля, вам нужно ее вызвать. Для этого можно просто вызвать функцию `Add`, возвращенную из вызова `cwrap`, так же, как и любую другую функцию (не используя массив). Замените код в функции `callAdd` кодом, показанным ниже:

```
function callAdd() {
  const add = Module.cwrap('Add',
      'number',
      ['number', 'number']);

  const result = add(4, 5);
  console.log(`Result: ${result}`);
}
```

Возвращаемое значение cwrap — JavaScript-функция

Вызывает функцию JavaScript, передавая значения напрямую

После того как изменения в функцию `callAdd` будут внесены, можно открыть браузер и ввести `http://localhost:8080/add.html` в адресную строку, чтобы увидеть только что настроенную веб-страницу. Если вы нажмете кнопку `Add` (Добавить),

то увидите результат вызова `Add` в окне консоли инструментов разработчика браузера, как показано на рис. Б.2.

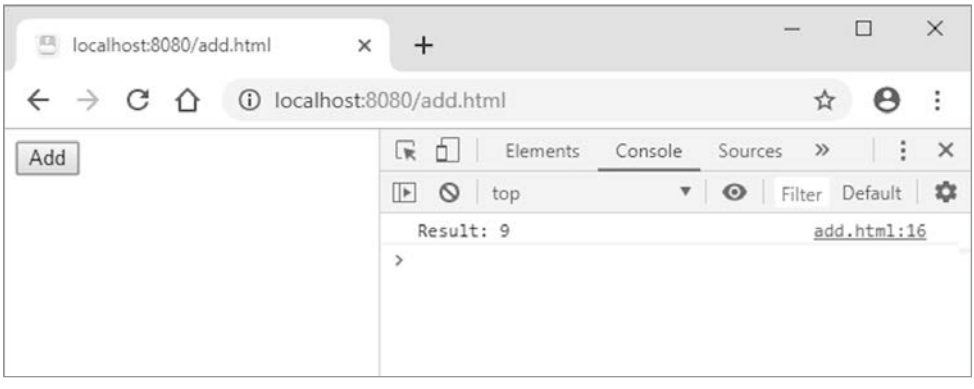


Рис. Б.2. Результат вызова функции Add модуля с помощью cwrap и передачи значений параметров 4 и 5

Б.3. ВЫЗОВЫ ФУНКЦИЙ НАПРЯМУЮ

Функции `ccall` и `cwrap` в Emscripten обычно используются при вызове функции в модуле, поскольку помогают при управлении памятью строк, если не обязательно, чтобы строка была долгоживущей.

Можно вызвать функцию модуля напрямую, но это значит, что ваш код должен будет обрабатывать все необходимое управление памятью. Если ваш код уже выполняет все это или в вызовах используются только числа с плавающей запятой и целые числа, которые не требуют управления памятью, то вам стоит рассмотреть данный подход.

Когда компилятор Emscripten создает модуль WebAssembly, он помещает символ подчеркивания перед именами функций. Важно помнить о следующих отличиях:

- при вызове `ccall` или `cwrap` символ подчеркивания не добавляется;
- при вызове функции напрямую необходимо добавить символ подчеркивания.

В следующем фрагменте кода показано, как напрямую вызвать функцию `Add` в модуле:

```
function callAdd() {
  const result = Module._Add(2, 5);
  console.log(`Result: ${result}`);
}
```

Вызов функции Add напрямую. Не забывайте добавить символ подчеркивания

Б.4. ПЕРЕДАЧА МАССИВА В МОДУЛЬ

Функции `ccall` и `cwrap` принимают тип `'array'`, но автоматическое управление памятью предназначено только для 8-битных значений. Если ваша функция ожидает, например, массив с целыми числами, то необходимо самостоятельно управлять памятью, выделив достаточно памяти для каждого элемента в массиве, скопировав содержимое массива в память модуля, а затем освободив память, после того как вернется результат вызова.

Память модуля WebAssembly — это просто буфер типизированного массива. Emscripten включает несколько представлений, которые позволяют просматривать память по-разному, что упрощает работу с разными типами данных:

- `HEAP8` — 8-битная знаковая память с использованием объекта `Int8Array` в JavaScript;
- `HEAP16` — 16-битная знаковая память с использованием объекта `Int16Array` в JavaScript;
- `HEAP32` — 32-битная знаковая память с использованием объекта `Int32Array` в JavaScript;
- `HEAPU8` — 8-битная беззнаковая память с использованием объекта `Uint8Array` в JavaScript;
- `HEAPU16` —16-битная беззнаковая память с использованием объекта `Uint16Array` в JavaScript;
- `HEAPU32` — 32-битная беззнаковая память с использованием объекта `Uint32Array` в JavaScript;
- `HEAPF32`— 32-битная память с плавающей запятой с использованием объекта `Float32Array` в JavaScript;
- `HEAPF64`— 64-битная память с плавающей запятой с использованием объекта `Float64Array` в JavaScript.

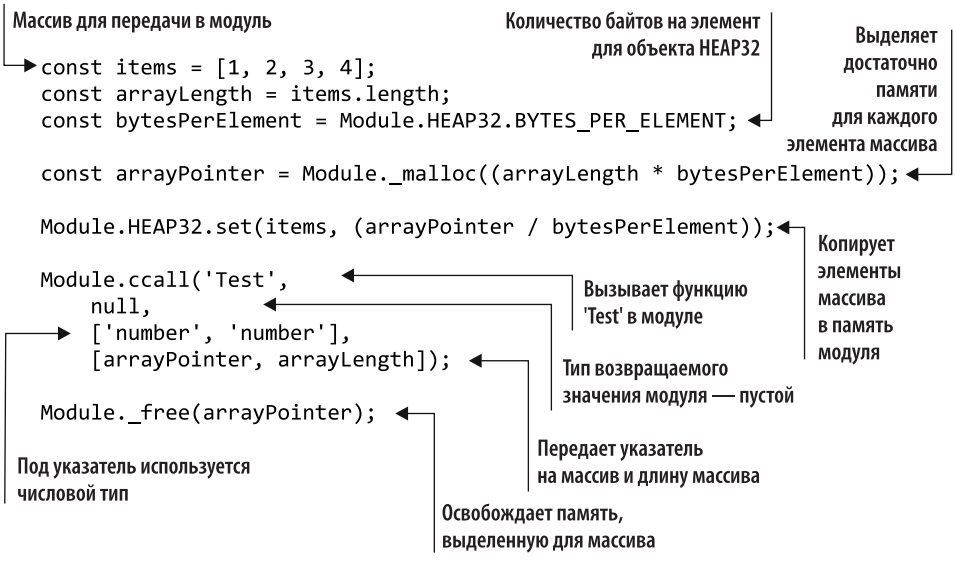
Например, если у вас есть массив целых чисел, то вы должны использовать представление `HEAP32`, которое в действительности является объектом `Int32Array` в JavaScript. Чтобы выделить достаточно памяти для указателя массива, вызовите `Module._malloc`, передав значение — результат умножения количества элементов в массиве на количество байтов для каждого элемента. Объект `Module.HEAP32` — это объект для 32-битных целых чисел, вследствие чего стоит использовать константу `Module.HEAP32.BYTES_PER_ELEMENT`, которая содержит значение `4`. Каждый объект кучи имеет константу `BYTES_PER_ELEMENT`.

Если у вас есть память, выделенная для указателя массива, то вы можете использовать функцию `set` объекта `HEAP32`. Первый параметр функции `set` — массив,

который нужно скопировать в память модуля WebAssembly. Второй параметр — индекс, с которого функция `set` должна начать запись данных в базовый массив (память модуля). В этом случае, поскольку вы работаете с 32-битным представлением памяти, каждый индекс относится к одной из групп по 32 бита (4 байта). В результате нужно разделить адрес памяти на 4. Можно задействовать стандартное деление, но также вы можете увидеть использование оператора побитового сдвига вправо в некоторых фрагментах кода — как в связующем коде Emscripten в JavaScript. Следующий пример будет означать то же, что и операция деления на 4, но будет применять оператор побитового сдвига вправо `arrayPointer >> 2`.

В листинге Б.2 показано, как ваш JavaScript передаст в модуль массив целых чисел.

Листинг Б.2. JavaScript передает массив целых чисел в модуль



B Макросы в Emscripten

В этом приложении

- ✓ Обзор серии макросов `emscripten_run_script`.
- ✓ Макрос `EM_JS` в Emscripten.
- ✓ Серия макросов `EM_ASM` в Emscripten.

Emscripten предоставляет три типа макросов, которые могут помочь вам взаимодействовать с хостом и могут пригодиться, когда вам нужно сделать такие вещи, как отладка проблем с кодом. Макросы в Emscripten бывают двух видов. Первый тип макроса — серия `emscripten_run_script`, а все другие типы — серии макросов `EM_JS` и `EM_ASM`.

B.1. МАКРОСЫ EMSCRIPTEN_RUN_SCRIPT

Серия макросов `emscripten_run_script` выполняет код JavaScript напрямую с помощью функции `eval` в JavaScript. Она представляет собой специальную функцию JavaScript, которая принимает строку и превращает ее в код JavaScript. Использование `eval` в JavaScript обычно не одобряется — это медленнее по сравнению с альтернативами. И что более важно: если строка, которую вы передаете, содержит данные, предоставленные пользователем, то эти данные превращаются

в код, который может делать что угодно, а это представляет серьезную угрозу безопасности. Другой недостаток применения функции `eval` заключается в том, что в зависимости от настроек безопасности браузер может вообще препятствовать работе `eval`, и ваш код может работать не так, как ожидалось.

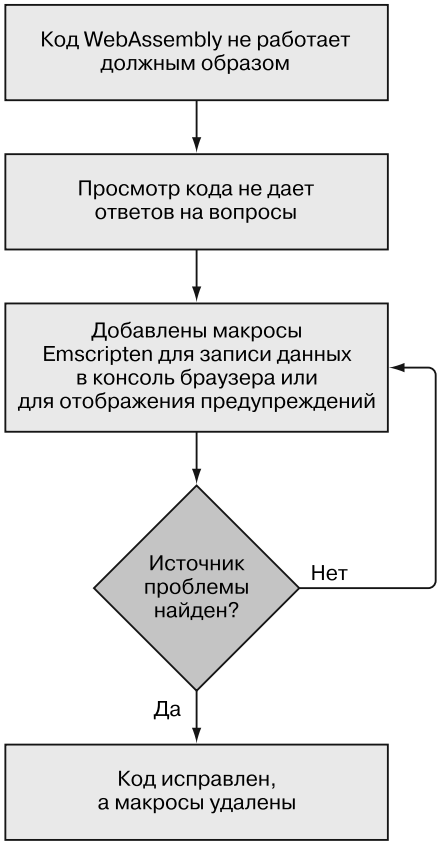


Рис. B.1. Отладка модуля WebAssembly с помощью макросов

Рекомендуется никогда не использовать серию макросов `emscripten_run_script` в производственном коде и особенно с данными, предоставленными пользователем. Однако эти макросы могут пригодиться при отладке. Например, как показано на рис. B.1, если модуль WebAssembly не работает должным образом и просмотр кода не помогает сузить радиус вероятных причин, то вы можете добавить макросы в определенные места кода. Возможно, вы начнете с добавления одного макроса для каждой функции, чтобы попытаться сузить радиус проблемы, отображая предупреждение или консольное сообщение. Кроме того, можно

добавить дополнительные макросы, чтобы еще больше сузить радиус проблемы, а затем, как только проблема будет обнаружена и исправлена, удалить макросы.

Макрос `emscripten_run_script` принимает указатель `const char*` и возвращает значение `void`. Ниже приводится пример использования `emscripten_run_script` для записи строки в консоль:

```
emscripten_run_script("console.log('The Test function')");
```

Макросы `emscripten_run_script_int` и `emscripten_run_script_string` также принимают указатель `const char*`, но разница между этими двумя типами заключается в их типах возвращаемого значения:

- `emscripten_run_script_int` возвращает целое число;
- `emscripten_run_script_string` возвращает указатель `char*`.

B.2. МАКРОСЫ EM_JS

Второй тип макросов Emscripten, доступный для модулей WebAssembly, — серии `EM_JS` и `EM_ASM`. Макрос `EM_JS` предлагает способ объявления функций JavaScript прямо в коде на C или C++, тогда как макросы `EM_ASM` позволяют использовать встроенный JavaScript.

Хотя код JavaScript для всех этих макросов находится внутри вашего кода на C или C++, компилятор Emscripten фактически создает необходимые функции JavaScript и скрыто вызывает эти функции, когда модуль запущен. В данном разделе вы сосредоточитесь на макросе `EM_JS`; макросы `EM_ASM` рассматриваются в следующем разделе.

Макрос `EM_JS` принимает четыре параметра:

- тип возвращаемого значения функции;
- название функции;
- аргументы функции, заключенные в круглые скобки. Если аргументов для передачи функции нет, то по-прежнему необходимы пустые открывающие и закрывающие круглые скобки;
- код тела функции.

ПРЕДУПРЕЖДЕНИЕ

При использовании этого макроса следует помнить о том, что первые три параметра записываются с помощью синтаксиса C++. Четвертый параметр, тело функции, представляет собой код JavaScript.

B.2.1. Без параметров

Первый макрос `EM_JS`, который вы определите, — функция JavaScript, не имеющая параметров и не возвращающая значение. Для начала нужно создать папку Appendix C\C.2.1 EM_JS\ для ваших файлов. Затем создайте в папке файл `em_js.c` и откройте его в своем любимом редакторе.

Макрос не требует, чтобы значение возвращалось функцией, поэтому укажите `void` для первого параметра. Имя макроса — `NoReturnValueWithNoParameters`, и поскольку никаких параметров не будет, в качестве третьего параметра макроса будут передаваться просто открывающие и закрывающие круглые скобки. Сам код JavaScript — это вызов `console.log` для отправки сообщения в окно консоли инструментов разработчика браузера, где указано, что макрос был вызван.

После того как макрос определен, вызов функции осуществляется так же, как и вызов обычной функции на C или C++. Вызов функции помещается в функцию `main`, чтобы код запускался автоматически при загрузке и создании экземпляра модуля. Добавьте следующий фрагмент кода в файл `em_js.c`:

```
#include <emscripten.h>

EM_JS(void, NoReturnValueWithNoParameters, (), {
  console.log("NoReturnValueWithNoParameters called");
});

int main() {
  NoReturnValueWithNoParameters();
  return 0;
}
```

Объявляет макрос

Записывает сообщение в консоль инструментов разработчика браузера

Вызывает функцию JavaScript, определенную с помощью макроса EM_JS

Нет необходимости проходить процесс создания простой HTML-страницы только для того, чтобы увидеть результаты действия макросов в данном приложении. Вместо этого скомпилируем созданный код в модуль WebAssembly и воспользуемся шаблоном HTML в Emscripten.

Чтобы скомпилировать только что написанный код, откройте командную строку, перейдите в папку, в которой сохранен файл `em_js.c`, и выполните следующую команду:

```
emcc em_js.c -o em_js.html
```

СПРАВКА

Вы можете увидеть предупреждающее сообщение о том, что для функции макроса не указаны аргументы. Его можно проигнорировать.

Создав файл WebAssembly, вы можете открыть браузер и ввести `http://localhost:8080/em_js.html` в адресную строку, чтобы увидеть веб-страницу. Если

вы откроете инструменты разработчика браузера, нажав клавишу F12, то увидите текст `NoReturnValueWithNoParameters called`, записанный в окно консоли, как показано на рис. B.2.

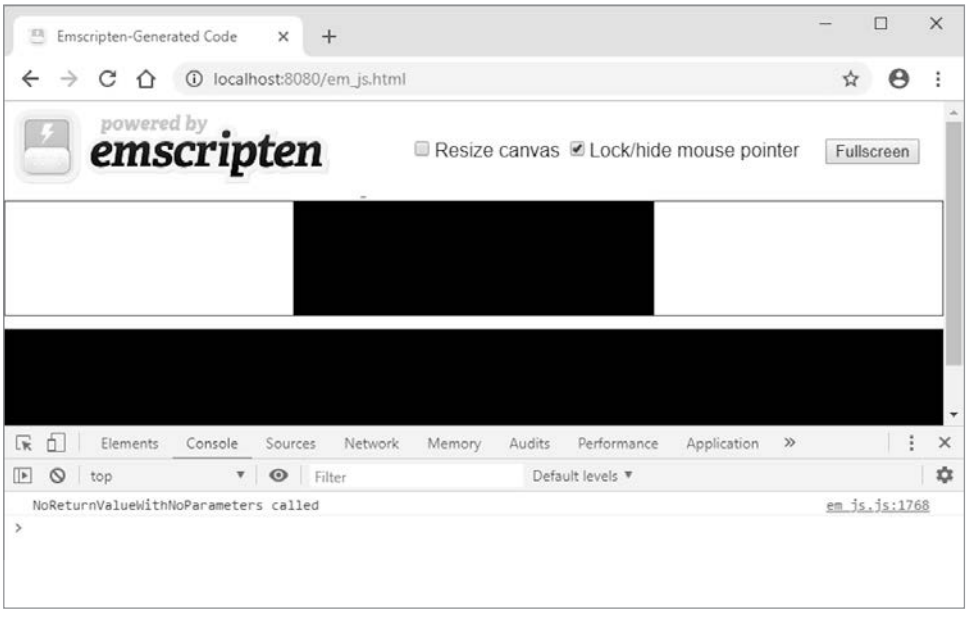


Рис. B.2. Вывод в окно консоли из макроса EM_JS NoReturnValueWithNoParameters

B.2.2. Передача значений параметров

В этом примере вы узнаете, как передавать значения в макрос `EM_JS` и как код JavaScript внутри взаимодействует с параметрами. В папке Appendix C\ создайте новую папку C.2.2 EM_JS\, а затем создайте в папке файл с именем `em_js.c`. Откройте файл в любимом редакторе.

Макрос не возвращает значение, поэтому установим для первого параметра значение `void`. Новый макрос будет называться `NoReturnValueWithIntegerAndDoubleParameters`, поскольку функция получит в качестве параметров `int` и `double`. Код JavaScript просто вызовет `console.log` для отображения в окне консоли сообщения, в котором говорится, что функция была вызвана, и отображаются переданные значения.

Создадим функцию `main`, которая будет вызываться автоматически при создании экземпляра модуля. В функции `main` вы вызываете свой макрос, передавая `integer` и `double`, так же, как и при вызове обычной функции.

Добавьте следующий фрагмент кода в файл `em_js.c`:

```
#include <emscripten.h>

EM_JS(void, NoReturnValueWithIntegerAndDoubleParameters,
    (int integer_value, double double_value), {
  console.log("NoReturnValueWithIntegerAndDoubleParameters
  ➥ called...integer_value: " +
      integer_value.toString() + " double_value: " +
      double_value.toString());
});

int main() {
  NoReturnValueWithIntegerAndDoubleParameters(1, 5.49);
  return 0;
}
```

Макрос имеет два параметра: int и double

Чтобы скомпилировать код, откройте командную строку, перейдите в папку, в которой сохранен файл `em_js.c`, а затем выполните следующую команду:

```
emcc em_js.c -o em_js.html
```

Создав файл WebAssembly, вы можете открыть браузер и ввести `http://localhost:8080/em_js.html` в адресную строку, чтобы увидеть веб-страницу. В окне консоли браузера вы должны увидеть текст, указывающий, что была вызвана функция `NoReturnValueWithIntegerAndDoubleParameters`, как показано на рис. В.3.

Рис. В.3. Вывод в окно консоли из макроса NoReturnValueWithIntegerAndDoubleParameters

B.2.3. Передача указателей в качестве параметров

Указатели также могут быть переданы как параметры в макрос `EM_JS`. Однако следует помнить, что код WebAssembly работает только с целыми типами данных и типами с плавающей точкой. Все остальные типы, например строки, помещаются в линейную память модуля. Хотя в вашем коде на C или C++ будет казаться, что вы передаете строковый литерал в функцию при компиляции модуля, код WebAssembly теперь будет указывать на место в памяти и передавать его функции.

В папке `Appendix C\` создайте новую папку `C.2.3 EM_JS\`, а затем добавьте файл `em_js.c`. Откройте файл в любимом редакторе.

Макрос не возвращает значение, будет иметь имя `NoReturnValueWithStringParameter` и примет в качестве параметра `const char*`. В нашем случае используется функция `console.log` для отправки сообщения в окно консоли браузера, в котором говорится, что макрос был вызван, и отображается полученное строковое значение. Поскольку строка будет находиться в памяти модуля, задействуем вспомогательную функцию `UTF8ToString` в Emscripten для чтения строки из памяти. Добавьте следующий фрагмент кода в файл `em_js.c`:

```
#include <emscripten.h>

EM_JS(void, NoReturnValueWithStringParameter,
    (const char* string_pointer), {
  console.log("NoReturnValueWithStringParameter called: " +
      Module.UTF8ToString(string_pointer));
});

int main() {
  NoReturnValueWithStringParameter("Hello from WebAssembly");
  return 0;
}
```

Макрос принимает const char* в качестве параметра

Считывает строку из памяти модуля

Поскольку для кода JavaScript потребуется вспомогательная функция `UTF8ToString` Emscripten, нужно будет включить эту функцию в флаг командной строки массива `EXTRA_EXPORTED_RUNTIME_METHODS` при создании модуля WebAssembly. Ниже приводится командная строка для компиляции кода:

```
emcc em_js.c -s EXTRA_EXPORTED_RUNTIME_METHODS=['UTF8ToString']
➥ -o em_js.html
```

Вы можете просмотреть веб-страницу в браузере, введя `http://localhost:8080/em_js.html` в адресную строку. В окне консоли браузера вы должны увидеть текст, указывающий, что была вызвана функция `NoReturnValueWithStringParameter` и что она получила текст `Hello from WebAssembly`, как показано на рис. B.4.

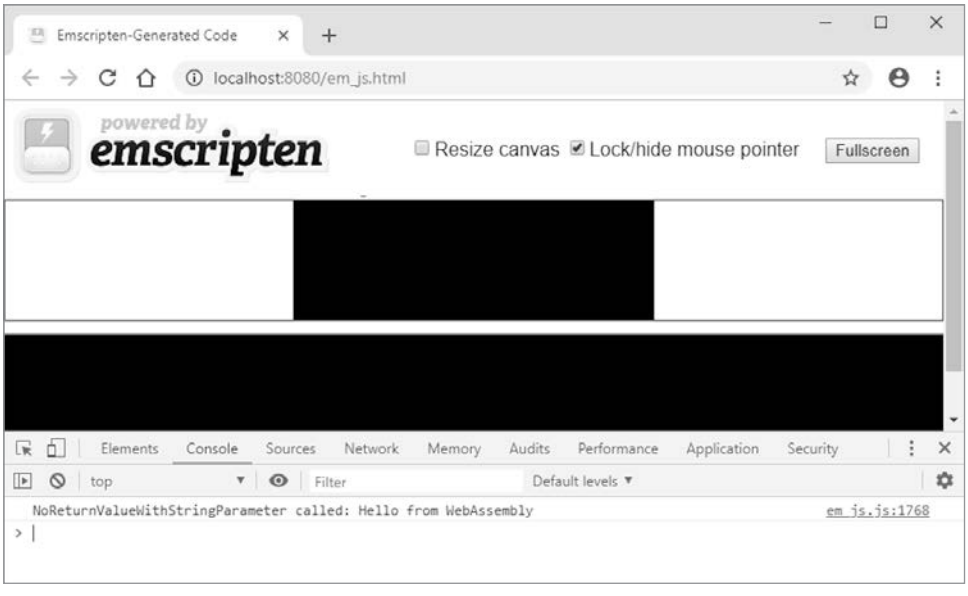


Рис. В.4. Вывод в окно консоли, показывающий, что был вызван макрос NoReturnValueWithStringParameter

В.2.4. Возвращение указателя на строку

Ни один из созданных до этого примеров `EM_JS` не вернул значение. Возвращать значения из функций `EM_JS` можно, но, как и в случае с параметрами, следует помнить, что код WebAssembly работает только с целыми типами данных и типами с плавающей запятой. Все другие типы, такие как строки, необходимо поместить в линейную память модуля.

В папке Appendix C \ создайте новую папку C.2.4 EM_JS\, а затем создайте в папке файл `em_js.c`. Откройте его в редакторе.

В этом примере мы определим функцию `StringReturnValueWithNoParameters`, которая не будет иметь параметров и станет возвращать указатель `char*`. В коде JavaScript определим строковую переменную с сообщением для возврата значения в модуль.

Чтобы передать строку в модуль, нужно определить, сколько байтов она содержит; для этого воспользуемся вспомогательной функцией `lengthBytesUTF8` в Emscripten. Как только вы узнаете количество байтов в строке, вы дадите модулю указание выделить часть своей памяти для строки с помощью функции `malloc` стандартной библиотеки C. Затем скопируете строку в память модуля с помощью

вспомогательной функции `stringToUTF8` в Emscripten. Наконец, код JavaScript вернет указатель на строку.

В функции `main` модуля при вызове макроса вы получите указатель на строку. Затем передадите строковый указатель функции `printf`, чтобы связующий код Emscripten записал сообщение в окно консоли инструментов разработчика браузера, а также в текстовое поле на веб-странице.

ПРИМЕЧАНИЕ

Главное, о чем следует помнить, — это то, что при использовании malloc вам нужно обязательно освободить память, иначе произойдет утечка памяти. Память можно освободить с помощью функции стандартной библиотеки C.

Поместите содержимое листинга B.1 в файл `em_js.c`.

Листинг B.1. Макрос EM_JS, который возвращает строку (em_js.c)

```
#include <stdlib.h>
#include <stdio.h>
#include <emscripten.h>

EM_JS(char*, StringReturnValueWithNoParameters, (), {
  const greetings = "Hello from StringReturnValueWithNoParameters";
  const byteCount = (Module.lengthBytesUTF8(greetings) + 1);

  const greetingsPointer = Module._malloc(byteCount);
  Module.stringToUTF8(greetings, greetingsPointer, byteCount);

  return greetingsPointer;
});

int main() {
  char* greetingsPointer = StringReturnValueWithNoParameters();

  printf("StringReturnValueWithNoParameters was called and it returned the
  ➥ following result: %s\n", greetingsPointer);

  free(greetingsPointer);

  return 0;
}
```

Строка, возвращаемая в модуль

Определяет макрос, который возвращает char*

Определяет количество байтов в строке; добавляет байт для символа конца строки

Выделяет часть памяти модуля для строки

Возвращает указатель на место строки в памяти модуля

Копирует строку в память модуля

Освобождает память, выделенную для указателя строки

Отображает строку в окне консоли браузера на веб-странице

Вызывает функцию JavaScript и получает указатель на строку

Код JavaScript будет использовать функции `lengthBytesUTF8` и `stringToUTF8`, поэтому необходимо включить их в массив командной строки `EXTRA_EXPORTED_RUNTIME_METHODS`. Ниже приводится командная строка для компиляции вашего кода в модуль WebAssembly:

```
emcc em_js.c -s EXTRA_EXPORTED_RUNTIME_METHODS=['lengthBytesUTF8',
➥'stringToUTF8'] -o em_js.html
```

СПРАВКА

Вы можете увидеть предупреждающее сообщение о том, что для функции макроса не указаны аргументы. Его можно проигнорировать.

Чтобы просмотреть веб-страницу в браузере, введите `http://localhost:8080/em_js.html` в адресную строку. Вы должны увидеть текст, указывающий на то, что была вызвана функция `StringReturnValueWithNoParameters` и что она получила текст `Hello from StringReturnValueWithNoParameters`, как показано на рис. B.5.

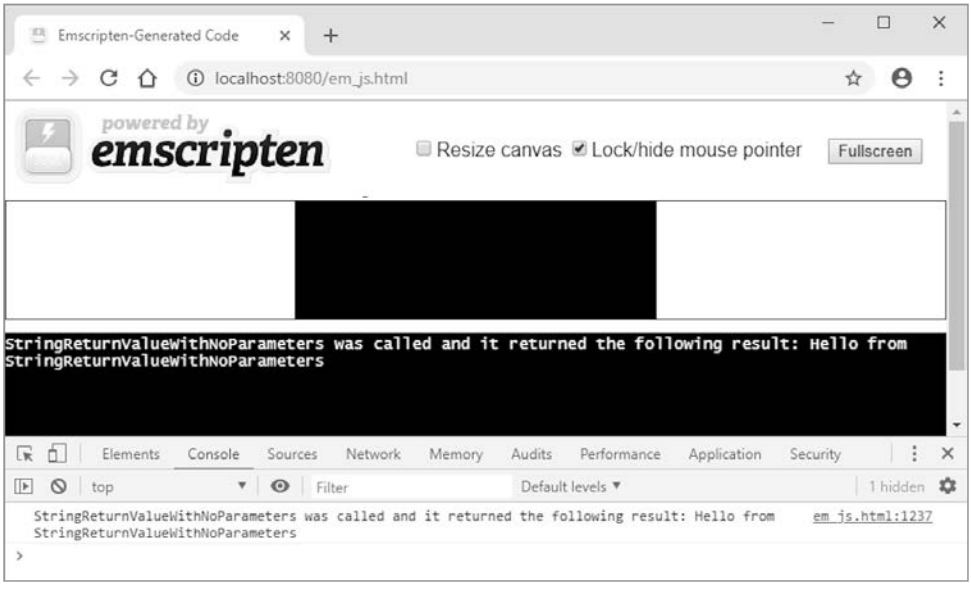


Рис. B.5. Вывод в окно консоли, показывающий, что был вызван макрос StringReturnValueWithNoParameters

B.3. МАКРОСЫ EM_ASM

Как упоминалось в предыдущем разделе, макрос `EM_JS` предлагает способ объявления функций JavaScript прямо в коде на C или C++. При использовании макросов `EM_ASM` вы не объявляете функцию JavaScript явно. Вместо этого вы добавляете встроенный JavaScript в свой код на C. Как в случае с макросами

EM_JS, так и с EM_ASM код JavaScript в действительности не находится в коде на C. Компилятор Emscripten фактически создает необходимые функции JavaScript и вызывает их «за кулисами» во время работы модуля.

Доступно несколько вариантов макроса EM_ASM:

- EM_ASM;
- EM_ASM_;
- EM_ASM_INT;
- EM_ASM_DOUBLE.

Макросы EM_ASM и EM_ASM_ не возвращают значение. Макрос EM_ASM_INT возвращает целое число, а макрос EM_ASM_DOUBLE — значение типа double.

B.3.1. EM_ASM

Макросы EM_ASM используются для выполнения JavaScript, указанного в открывающих и закрывающих скобках макроса. Чтобы увидеть это, в папке Appendix C\ создайте папку C.3.1 EM_ASM\, а затем создайте в папке файл em_asm.c. Откройте файл в редакторе.

Создайте функцию main и добавьте вызов макроса EM_ASM, чтобы просто записать строку в консоль инструментов разработчика браузера. Добавьте следующий фрагмент кода в файл em_asm.c:

```
#include <emscripten.h>

int main() {
  EM_ASM(console.log('EM_ASM macro calling'));
}
```

Можно дать Emscripten указание скомпилировать код в модуль WebAssembly и сгенерировать HTML-шаблон, открыв командную строку, перейдя туда, где сохранен файл em_asm.c, а затем запустив следующую команду:

```
emcc em_asm.c -o em_asm.html
```

Вы можете просмотреть веб-страницу в браузере, введя http://localhost:8080/em_asm.html в адресную строку. В окне консоли браузера должен появиться текст EM_ASM macro calling, как показано на рис. B.6.

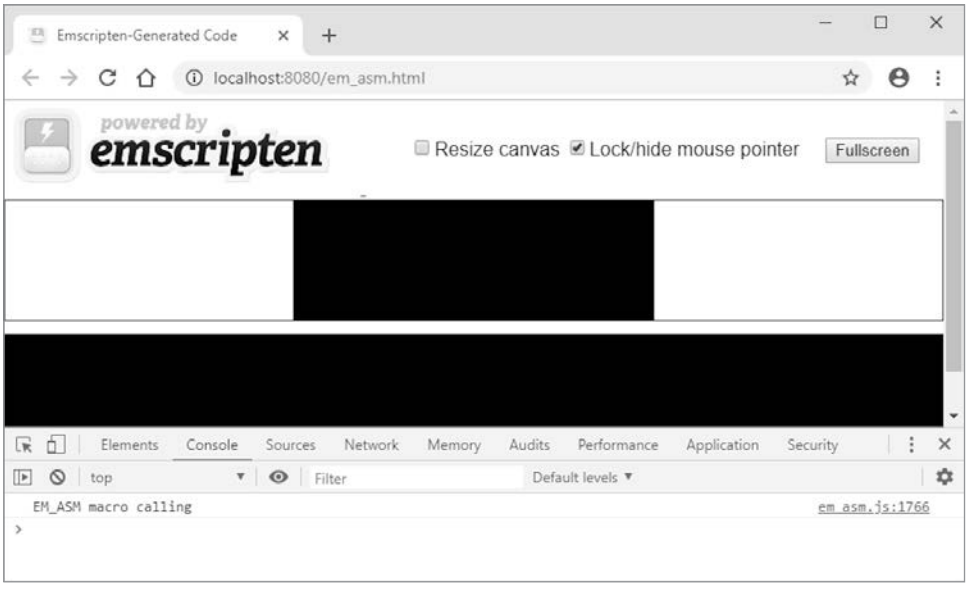


Рис. В.6. Вывод в окно консоли при вызове функции EM_ASM

В.3.2. EM_ASM_

Макрос `EM_ASM_` используется для передачи одного или нескольких значений из кода на C или C++ в код JavaScript, определенный в макросе. Хотя показанный ранее макрос EM_ASM также может применяться для передачи значений в код JavaScript, который он содержит, рекомендуется вместо этого обращаться к макросу `EM_ASM_`. Преимущество заключается в том, что, если разработчик забудет передать значение, то компилятор выдаст ошибку.

Первый параметр макросов `EM_ASM` и `EM_ASM_` содержит код JavaScript, а любые дополнительные параметры — это значения, передаваемые из кода на C или C++ в код JavaScript внутри макроса:

- каждый переданный параметр будет рассматриваться кодом JavaScript как `$0`, `$1`, `$2` и т. д.;
- каждый параметр, передаваемый в макрос, может быть только `int32_t` или `double`, но указатели в WebAssembly являются 32-битными целыми числами, поэтому их также можно передавать.

Фигурные скобки вокруг кода JavaScript в макросах `EM_ASM` не требуются, но они помогают различать код JavaScript и передаваемые значения на C или C++.

В папке Appendix C\ создайте папку C.3.2 EM_ASM_\, а затем добавьте файл `em_asm.c`. Откройте файл в редакторе.

Теперь создадим функцию `main`, а внутри функции вызовем макрос `EM_ASM_`, передав целочисленное значение `10`. JavaScript в макросе просто выведет сообщение в консоль браузера, указав полученное значение. Добавьте следующий фрагмент кода в файл `em_asm_.c`:

```
#include <emscripten.h>

int main() {
  EM_ASM_({
    console.log('EM_ASM_ macro received the value: ' + $0);
  }, 10);
}
```

Значения принимаются как переменные $0, $1, $2 и т. д.

В код JavaScript можно передавать только значения int32_t или double в C/C++

Чтобы создать модуль WebAssembly, откройте окно консоли, перейдите в папку, в которой находится ваш файл `em_asm_.c`, а затем выполните следующую команду:

```
emcc em_asm_.c -o em_asm_.html
```

Как показано на рис. B.7, если вы введете `http://localhost:8080/em_asm_.html` в адресную строку браузера, то увидите текст, указывающий, что макрос `EM_ASM_` получил значение `10`.

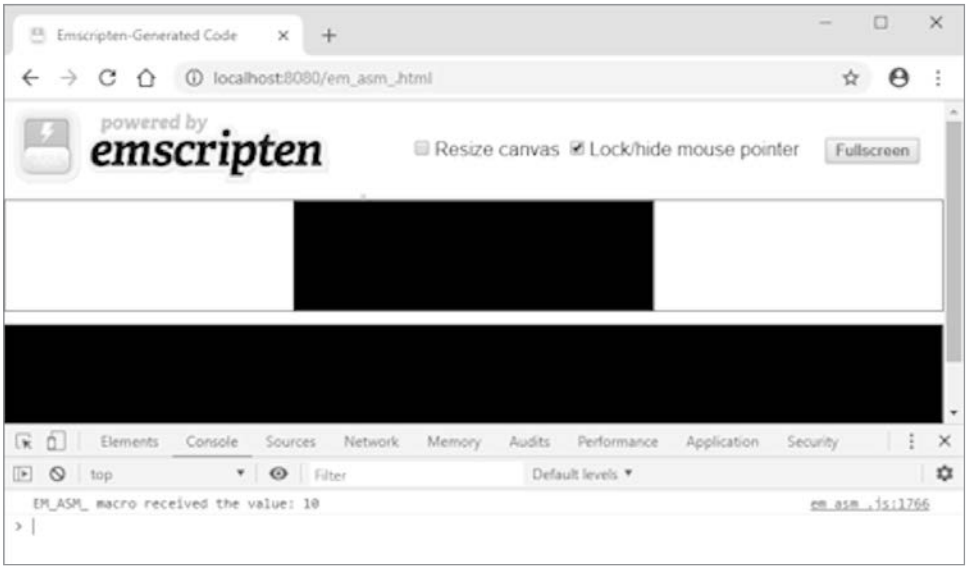


Рис. B.7. Вывод в окно консоли при вызове функции EM_ASM_

B.3.3. Передача указателей в качестве параметров

В этом примере мы собираемся передать строку в код JavaScript макроса `EM_ASM_`. Единственные типы данных, которые поддерживают модули WebAssembly, — целые числа и числа с плавающей запятой. Любой другой тип данных, например строки, должен быть представлен в линейной памяти модуля.

Прежде чем начать, нужно создать папку C.3.3 EM_ASM_ \ в вашей папке Appendix C\, а затем добавить файл `em_asm_.c`. Откройте его в редакторе.

Необходимо создать функцию `main`. В функции `main` вы вызовете макрос `EM_ASM_`, передающий строковый литерал `"world!"`. Поскольку модули WebAssembly поддерживают только целые числа и числа с плавающей запятой, когда код компилируется в модуль WebAssembly, строка `"world!"` фактически будет помещена в линейную память модуля. Указатель будет передан на код JavaScript внутри макроса, поэтому нужно будет использовать вспомогательную функцию `UTF8ToString` в Emscripten для чтения строки из памяти модуля, прежде чем можно будет записать строку в окно консоли инструментов разработчика браузера. Добавьте следующий фрагмент кода в файл `em_asm_.c`:

```
#include <emscripten.h>

int main() {
  EM_ASM_({
    console.log('hello ' + Module.UTF8ToString($0));
  }, "world!");
}
```

Считывает строку из памяти модуля

Строка передается в код JavaScript как указатель

Поскольку в коде JavaScript будет использоваться вспомогательная функция `UTF8ToString` в Emscripten, нужно будет включить ее в массив командной строки `EXTRA_EXPORTED_RUNTIME_METHODS` при создании модуля WebAssembly. Ниже приводится командная строка для компиляции кода:

```
emcc em_asm_.c -s EXTRA_EXPORTED_RUNTIME_METHODS=['UTF8ToString']
➥ -o em_asm_.html
```

Введите `http://localhost:8080/em_asm_.html` в адресную строку браузера, чтобы увидеть готовую веб-страницу. Как показано на рис. B.8, в окне консоли инструментов разработчика браузера вы должны увидеть текст `hello world!`.

B.3.4. EM_ASM_INT и EM_ASM_DOUBLE

Бывают случаи, когда нужно вызвать JavaScript, чтобы запросить значение. Для этого применим либо макрос `EM_ASM_INT`, который возвращает целое число, либо макрос `EM_ASM_DOUBLE`, который возвращает значение с плавающей запятой.

Рис. B.8. Вывод в окно консоли при вызове функции EM_ASM_

Как и в случае с макросом `EM_ASM_`, необязательные значения могут быть переданы из кода на C или C++ в код JavaScript. В этом примере вы вызовете макрос `EM_ASM_DOUBLE`, передав два значения `double` в качестве параметров. JavaScript просуммирует оба значения и вернет результат. Вы поместите код в функцию `main` и передадите результат макроса и кода JavaScript в Emscripten с помощью функции `printf`.

В папке Appendix C\ создайте папку C.3.4 EM_ASM_DOUBLE\. Создайте файл `em_asm_double.c` и откройте его в редакторе. Добавьте в файл следующий фрагмент кода:

```
#include <stdio.h>
#include <emscripten.h>

int main() {
  double sum = EM_ASM_DOUBLE({
    return $0 + $1;
  }, 10.5, 20.1);

  printf("EM_ASM_DOUBLE result: %.2f\n", sum);
}
```

Откройте командную строку, перейдите в папку, в которой сохранен файл `em_asm_double.c`, а затем выполните следующую команду, чтобы создать модуль WebAssembly:

```
emcc em_asm_double.c -o em_asm_double.html
```

Вы можете открыть браузер и ввести `http://localhost:8080/em_asm_double.html` в адресную строку, чтобы увидеть только что сгенерированную веб-страницу. В окне консоли инструментов разработчика браузера и в текстовом поле на веб-странице должен появиться текст `EM_ASM_DOUBLE result: 30.60` (рис. В.9).

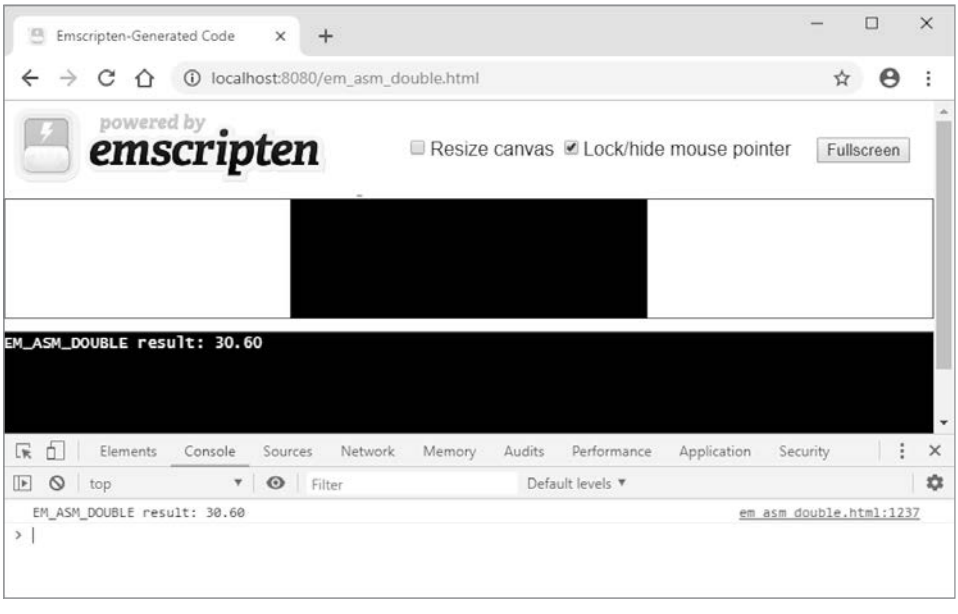


Рис. В.9. Результат 30.60 от вызова макроса EM_ASM_DOUBLE

В.3.5. Возврат указателя на строку

Можно вернуть строковый указатель из макроса `EM_ASM_INT`, поскольку указатели представлены в WebAssembly как 32-битные целые числа. Однако требуется управление памятью. Чтобы передать строку из кода JavaScript в модуль, эту строку необходимо скопировать в память модуля; затем указатель возвращается в модуль. Когда модуль закончит работу с указателем, ему необходимо освободить выделенную память.

В папке Appendix C\ создайте папку C.3.5 EM_ASM_INT\. Добавьте файл `em_asm_int.c` и откройте его в редакторе.

В JavaScript-коде макроса `EM_ASM_INT` вы определяете строку, а затем используете вспомогательную функцию `lengthBytesUTF8` Emscripten для определения количества

байтов в строке. Как только вы узнаете результат, можно дать модулю указание выделить необходимый объем линейной памяти для хранения строки. Чтобы выделить память, воспользуемся функцией `malloc` из стандартной библиотеки C. Последний шаг — копирование строки в память модуля с помощью вспомогательной функции `stringToUTF8` в Emscripten и затем возврат указателя коду на C.

Код будет помещен в функцию `main`, а результат вызова макроса `EM_ASM_INT` будет преобразован из целого числа в `char*`. Затем код передаст указатель на функцию `printf`, поэтому связующий код Emscripten будет показывать сообщение в окне консоли инструментов разработчика браузера, а также в текстовом поле веб-страницы. Перед завершением работы функции `main` выделенная память будет освобождена с помощью функции `free` стандартной библиотеки C:

```
#include <stdlib.h>
#include <stdio.h>
#include <emscripten.h>

int main() {
  char* message = (char*)EM_ASM_INT({
    const greetings = "Hello from EM_ASM_INT!";
    const byteCount = (Module.lengthBytesUTF8(greetings) + 1);

    const greetingsPointer = Module._malloc(byteCount);
    Module.stringToUTF8(greetings, greetingsPointer, byteCount);

    return greetingsPointer;
  });

  printf("%s\n", message);
  free(message);
}
```

Преобразует целочисленное возвращаемое значение в char*

Отображает сообщение в окне консоли браузера

Освобождает память, выделенную для указателя

Поскольку в коде JavaScript будут использоваться функции `lengthBytesUTF8` и `stringToUTF8`, необходимо добавить их в массив командной строки `EXTRA_EXPORTED_RUNTIME_METHODS`. Ниже приводится командная строка, необходимая для компиляции вашего кода в модуль WebAssembly:

```
emcc em_asm_int.c
➥ -s EXTRA_EXPORTED_RUNTIME_METHODS=['lengthBytesUTF8',
➥'stringToUTF8'] -o em_asm_int.html
```

Если вы откроете браузер и наберете `http://localhost:8080/em_asm_int.html` в адресной строке, то увидите только что созданную веб-страницу. В окне консоли браузера и в текстовом поле на веб-странице должен появиться текст `Hello from EM_ASM_INT!` (рис. B.10).

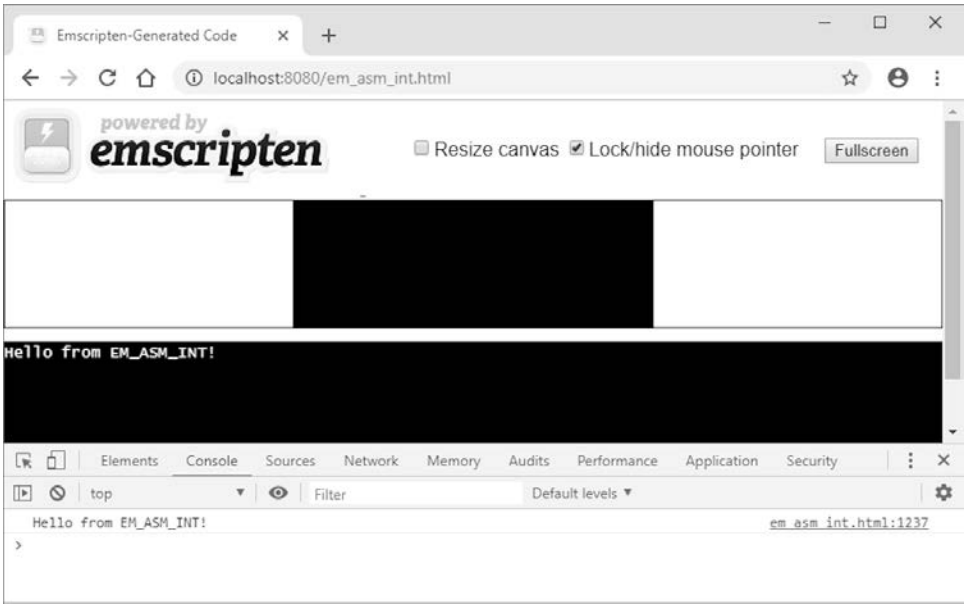


Рис. В.10. Сообщение макроса EM_ASM_INT, показанное в окне консоли инструментов разработчика браузера, а также в текстовом поле на веб-странице

Г Ответы к упражнениям

В этом приложении

✓ Ответы к упражнениям в конце глав.

Г.1. ГЛАВА 3

В главе 3 есть два упражнения.

Г.1.1. Упражнение 1

Какие четыре типа данных поддерживает WebAssembly?

Решение

32- и 64-битные целые числа, 32- и 64-битные числа с плавающей запятой.

Г.1.2. Упражнение 2

Добавьте функцию `Decrement` к вспомогательному модулю, созданному в подразделе 3.6.1:

а) функция должна иметь целочисленное возвращаемое значение и целочисленный параметр. Вычтите 1 из полученного значения и верните результат вызывающей функции;

б) скомпилируйте вспомогательный модуль, а затем измените JavaScript, чтобы вызвать функцию и отобразить результат в консоли.

Решение

В папке WebAssembly\ создайте папку Appendix D\D.1.2\source\. Скопируйте файл `side_module.c` из папки Chapter 3\3.6 side_module в новую папку source\.

Откройте файл `side_module.c` и добавьте функцию, показанную ниже, после функции `Increment`:

```
int Decrement(int value) {
  return (value - 1);
}
```

Чтобы скомпилировать код в модуль WebAssembly, перейдите в папку Appendix D\D.1.2\source\ и выполните следующую команду:

```
emcc side_module.c -s SIDE_MODULE=2 -O1
➥ -s EXPORTED_FUNCTIONS=['_Increment','_Decrement']
➥ -o side_module.wasm
```

В папке Appendix D\D.1.2\ создайте папку frontend\ и скопируйте в нее следующие файлы:

- `side_module.wasm` из папки source\;
- `side_module.html` из папки Chapter 3\3.6 side_module\.

Откройте файл `side_module.html` в редакторе. В методе `then` вызова `WebAssembly.instantiateStreaming` измените значение переменной с `const` на `let`. После вызова `console.log` добавьте вызов функции `_Decrement`, передав значение `4` и записав результат в консоль. Теперь код метода `then` должен соответствовать показанному ниже:

```
.then(result => {
  let value = result.instance.exports._Increment(17);
  console.log(value.toString());

  value = result.instance.exports._Decrement(4);
  console.log(value.toString());
});
```

Г.2. ГЛАВА 4

В главе 4 есть два упражнения.

Г.2.1. Упражнение 1

Какие два способа позволяют Emscripten сделать ваши функции видимыми для JavaScript-кода?

Решение

Существует два способа:

- включить в функцию объявление `EMSCRIPTEN_KEEPALIVE`;
- включить имена функций в массив `EXPORTED_FUNCTIONS` командной строки при компиляции модуля.

Г.2.2. Упражнение 2

Как предотвратить искажение имен функций при компиляции, чтобы код JavaScript мог использовать ожидаемое имя функции?

Решение

Использовать `extern "C"`.

Г.3. ГЛАВА 5

В главе 5 есть два упражнения.

Г.3.1. Упражнение 1

Какое ключевое слово нужно использовать для определения сигнатуры в вашем коде на C или C++, чтобы компилятор знал, что функция будет доступна при запуске кода?

Решение

`extern`

Г.3.2. Упражнение 2

Предположим, вам нужно добавить функцию в код JavaScript в Emscripten, которую ваш модуль будет вызывать, чтобы определять, подключено ли устройство пользователя к сети. Как бы вы добавили функцию `IsOnline`, которая возвращает `1`, если истина, и `0` (ноль), если ложь?

Решение

В коде C нужно определить функцию, как показано ниже:

```
extern int IsOnline();
```

При необходимости код на C вызывает функцию `IsOnline`, как и любую другую функцию. Например:

```
if (IsOnline() == 1) { /* request data from the server perhaps */ }
```

Чтобы добавить функцию JavaScript в код JavaScript, созданный Emscripten, используйте функцию `mergeInto`. У браузеров есть объект `navigator`, к которому можно получить доступ, чтобы определить, находится ли браузер в сети, с помощью метода `navigator.onLine`. Если хотите узнать об этом методе больше, то посетите следующую страницу MDN Web Docs: http://mng.bz/yzZe.

В файле JavaScript, который вы укажете в командной строке (`mergeinto.js`), у вас будет функция, подобная этой:

```
mergeInto(LibraryManager.library, {
  IsOnline: function() {
    return (navigator.onLine ? 1 : 0);
  }
});
```

В командной строке вы даете Emscripten указание включить вашу функцию в сгенерированный файл JavaScript, добавив параметр `--js-library`, за которым следует файл JavaScript с кодом `mergeInto`, как показано в следующем примере:

```
emcc test.cpp --js-library mergeinto.js -o test.html
```

Г.4. ГЛАВА 6

В главе 6 есть два упражнения.

Г.4.1. Упражнение 1

Какие две функции используются для добавления и удаления указателей функций из вспомогательного массива Emscripten?

Решение

`addFunction` и `removeFunction`

Г.4.2. Упражнение 2

Какую инструкцию WebAssembly применяет для вызова функции, определенной в разделе «Таблица»?

Решение

`call_indirect`

Г.5. ГЛАВА 7

В главе 7 есть два упражнения.

Г.5.1. Упражнение 1

Используя один из методов динамического связывания, которые вы изучили в этой главе, создайте следующее:

- вспомогательный модуль, содержащий функцию `Add`, которая принимает два целочисленных параметра и возвращает сумму как целое число;
- основной модуль, который имеет функцию `main()`, вызывающую функцию `Add` вспомогательного модуля, и отображает результат в окне консоли инструментов разработчика браузера.

Решение для вспомогательного модуля

В папке `WebAssembly\` создайте папку `Appendix D\D.5.1\source\`. В новой папке создайте файл `add.c`, а затем откройте его в своем любимом редакторе.

Поместите заголовочный файл для Emscripten и функцию `Add`, показанную ниже, в файл `add.c`:

```
#include <emscripten.h>

EMSCRIPTEN_KEEPALIVE
int Add(int value1, int value2) {
  return (value1 + value2);
}
```

В качестве альтернативы можно использовать массив командной строки EXPORTED_FUNCTIONS

Далее нужно будет скомпилировать файл `add.c` как вспомогательный модуль WebAssembly. Откройте командную строку, перейдите в папку Appendix D\D.5.1\source\, а затем выполните следующую команду:

```
emcc add.c -s SIDE_MODULE=2 -O1 -o add.wasm
```

Вторая часть упражнения — создать основной модуль, в котором есть функция `main`. Ручной подход для динамического связывания с использованием JavaScript API в WebAssembly может служить для связывания двух модулей, однако задействует два вспомогательных модуля. Два подхода, в которых применяются основные модули, — это `dlopen` и `dynamicLibraries`.

В функции `main` нужно вызвать функцию `Add` вспомогательного модуля, а затем отобразить результат в окне консоли инструментов разработчика браузера. Сначала посмотрим на подход с `dlopen`.

Решение для основного модуля: dlopen

В папке Appendix D\D.5.1\source\ создайте файл `main_dlopen.cpp`. Добавьте в файл код, показанный в листинге Г.1.

Листинг Г.1. Подход dlopen для основного модуля

```
#include <cstdlib>
#include <cstdio>
#include <dlfcn.h>
#include <emscripten.h>

typedef int(*Add)(int,int);

void CallAdd(const char* file_name) {
  void* handle = dlopen(file_name, RTLD_NOW);
  if (handle == NULL) { return; }

  Add add = (Add)dlsym(handle, "Add");
  if (add == NULL) { return; }

  int result = add(4, 9);

  dlclose(handle);

  printf("Result of the call to the Add function: %d\n", result);
}

int main() {
  emscripten_async_wget("add.wasm",
    "add.wasm",
    CallAdd,
    NULL);

  return 0;
}
```

Заголовочный файл для dlopen и связанных функций

Сигнатура функции для функции Add во вспомогательном модуле

Функция обратного вызова после завершения загрузки файла add.wasm

Открывает вспомогательный модуль

Получает ссылку на функцию Add

Вызывает функцию Add с помощью указателя функции

Закрывает вспомогательный модуль

Отображает результат выполнения функции Add в окне консоли браузера

Загружает файл add.wasm в файловую систему Emscripten

Называет загруженный файл add.wasm

Функция CallAdd будет вызвана при успешной загрузке

В случае сбоя загрузки функция обратного вызова не предусмотрена

Следующий шаг — компиляция файла `main_dlopen.cpp` как основного модуля WebAssembly, а Emscripten также сгенерирует файл шаблона HTML. Откройте командную строку, перейдите в папку Appendix D\D.5.1\source\, а затем выполните такую команду:

```
emcc main_dlopen.cpp -s MAIN_MODULE=1 -o main_dlopen.html
```

Если вы решили использовать подход `dynamicLibraries` для основного модуля, то посмотрим, как это можно сделать.

Решение для основного модуля: dynamicLibraries

Первый шаг в этом подходе — создание файла JavaScript, который будет содержать JavaScript для обновления свойства `dynamicLibraries` объекта `Module` в Emscripten. В папке Appendix D\D.5.1\source\ создайте файл `pre.js` и откройте его в редакторе. Добавьте код, показанный ниже, в файл `pre.js`, чтобы Emscripten связал `add.wasm` со вспомогательным модулем во время инициализации:

```
Module['dynamicLibraries'] = ['add.wasm'];
```

Второй шаг — создание C++ для основного модуля. В папке Appendix D\D.5.1\source\ создайте файл `main_dynamicLibraries.cpp` и откройте его в редакторе. Добавьте код, показанный в листинге Г.2, в этот файл.

Листинг Г.2. Подход dynamicLibraries для основного модуля

```
#include <cstdlib>
#include <cstdio>
#include <emscripten.h>

#ifdef __cplusplus
extern "C" {
#endif

extern int Add(int value1, int value2);

int main() {
  int result = Add(24, 76);
  printf("Result of the call to the Add function: %d\n", result);

  return 0;
}

#ifdef __cplusplus
}
#endif
```

Таким образом, компилятор знает, что функция будет доступна при запуске кода

Вызывает функцию Add

Отображает результаты в окне консоли браузера

Последний шаг — компиляция файла `main_dynamicLibraries.cpp` как основного модуля WebAssembly, а Emscripten также сгенерирует файл шаблона HTML.

Откройте командную строку, перейдите в папку Appendix D\D.5.1\source\, а затем выполните следующую команду:

```
emcc main_dynamicLibraries.cpp -s MAIN_MODULE=1
➥ --pre-js pre.js -o main_dynamicLibraries.html
```

Г.5.2. Упражнение 2

Какой подход динамического связывания вы бы использовали, если бы вам нужно было вызвать функцию во вспомогательном модуле, но она называлась бы так же, как функция в основном?

Решение

Подход `dlopen`.

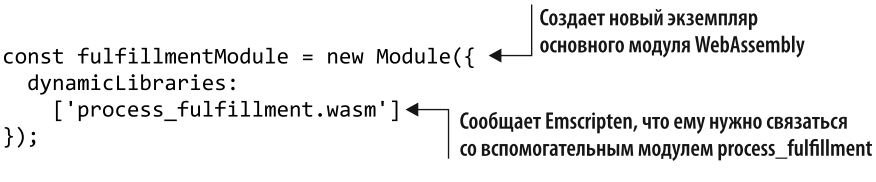
Г.6. ГЛАВА 8

В главе 8 есть два упражнения.

Г.6.1. Упражнение 1

Предположим, у вас есть вспомогательный модуль `process_fulfillment.wasm`. Как можно было бы создать новый экземпляр объекта `Module` в Emscripten и дать ему указание динамически связываться с этим вспомогательным модулем?

Решение



Г.6.2. Упражнение 2

Какой флаг необходимо передать Emscripten при компиляции основного модуля WebAssembly, чтобы объект `Module` был заключен в функцию в созданном Emscripten файле JavaScript?

Решение

```
-s MODULARIZE=1
```

Г.7. ГЛАВА 9

В главе 9 есть два упражнения.

Г.7.1. Упражнение 1

Если бы вы хотели задействовать функцию из стандарта C++17, то какой флаг применили бы при компиляции модуля WebAssembly, чтобы сообщить Clang о необходимости использования этого стандарта?

Решение

```
-std=c++17
```

Г.7.2. Упражнение 2

Протестируйте изменение логики `calculate_primes` из раздела 9.4, используя три потока, а не четыре, чтобы увидеть, как это повлияет на продолжительность вычислений. Протестируйте тот же код с помощью пяти потоков и поместите вычисление основного потока в pthread, чтобы увидеть, влияет ли перенос всех вычислений из основного потока на их продолжительность.

Решение для трех потоков

В папке WebAssembly\ создайте папку Appendix D\D.7.2\source\. Скопируйте файл `calculate_primes.cpp` из папки Chapter 9\9.4 pthreads\source\ в новую папку source\ и переименуйте его в `calculate_primes_three_pthreads.cpp`.

Откройте файл `calculate_primes_three_pthreads.cpp` в своем любимом редакторе. Внесите следующие изменения в функцию `main`:

- теперь массив `thread_ids` будет содержать три значения;
- теперь массив `args` будет содержать четыре значения;
- установите начальное значение `args_start`, равное `250000` (четверть от общего диапазона 1 000 000);
- цикл `pthread_create` должен выполниться, пока `i` меньше `3`;
- в цикле `pthread_create` значение `args [args_index].end` становится равным `args_start + 249999`. Значение `args_start` в конце цикла необходимо увеличить на `250000`;
- измените вызов `FindPrimes` для основного потока так, чтобы конечное значение (второй параметр) было равно `249999`;
- циклу `pthread_join` теперь нужно выполняться, пока `j` меньше `3`;

- наконец, цикл, который выводит найденные простые числа, будет выполняться, пока k меньше 4.

Теперь функция `main` должна соответствовать коду, показанному в листинге Г.3.

Листинг Г.3. Функция main в файле calculate_primes_three_pthreads.cpp

```
...

int main() {
  int start = 3, end = 1000000;
  printf("Prime numbers between %d and %d:\n", start, end);

  std::chrono::high_resolution_clock::time_point duration_start =
      std::chrono::high_resolution_clock::now();

  pthread_t thread_ids[3];  <-- Уменьшилось до 3
  struct thread_args args[4];  <-- Уменьшилось до 4

  int args_index = 1;
  int args_start = 250000;  <-- Диапазон первого потока начинается с 250 000

  for (int i = 0; i < 3; i++) {  <-- Уменьшилось на 3
    args[args_index].start = args_start;
    args[args_index].end = (args_start + 249999);  <-- Конец диапазона теперь равен 249 999 после значения args_start

    if (pthread_create(&thread_ids[i], NULL, thread_func,
        &args[args_index])) {
      perror("Thread create failed");
      return 1;
    }

    args_index += 1;
    args_start += 250000;  <-- Увеличивается на 250 000
  }

  FindPrimes(3, 249999, args[0].primes_found);  <-- Увеличивает конечное значение до 249 999

  for (int j = 0; j < 3; j++) {  <-- Уменьшилось до 3
    pthread_join(thread_ids[j], NULL);
  }

  std::chrono::high_resolution_clock::time_point duration_end =
      std::chrono::high_resolution_clock::now();

  std::chrono::duration<double, std::milli> duration =
      (duration_end - duration_start);

  printf("FindPrimes took %f milliseconds to execute\n", duration.count());

  printf("The values found:\n");
  for (int k = 0; k < 4; k++) {  <-- Уменьшилось до 4
```

```
    for(int n : args[k].primes_found) {
      printf("%d ", n);
    }
  }
  printf("\n");

  return 0;
}
```

Следующий шаг — компиляция файла `calculate_primes_three_pthreads.cpp`, а также выдача Emscripten команды сгенерировать файл шаблона HTML. Откройте командную строку, перейдите в папку Appendix D\D.7.2\source\ и выполните такую команду:

```
emcc calculate_primes_three_pthreads.cpp -O1 -std=c++11
➥ -s USE_PTHREADS=1 -s PTHREAD_POOL_SIZE=3
➥ -o three_pthreads.html
```

Итоги сравнения этих результатов с результатами из главы 9 и решением с пятью потоками приводятся после решения с пятью потоками.

Решение для пяти потоков

В папке Appendix D\D.7.2\source\ добавьте копию файла `calculate_primes_three_pthreads.cpp` и назовите ее `calculate_primes_five_pthreads.cpp`. Откройте файл с помощью вашего любимого редактора и внесите следующие изменения в функцию `main`:

- значение `start` теперь будет равно `0`;
- массивы `thread_ids` и `args` будут содержать пять значений;
- удалите строку кода `int args_index = 1`, а затем установите начальное значение `args_start`, равное `0`;
- циклу `pthread_create` нужно выполняться, пока `i` меньше `5`;
- в цикле `pthread_create`:
  - установите значение `args[args_index].end`, равное `args_start + 199999`;
  - значение `args_start` в конце цикла необходимо увеличить на `200000`;
  - удалите строку кода `args_index += 1` в конце цикла. В строках кода цикла `args [args_index]` замените `args_index` на `i`;
- удалите вызов `FindPrimes` из основного потока (непосредственно перед циклом `pthread_join`);
- циклу `pthread_join` нужно выполняться, пока `j` меньше `5`;
- наконец, цикл, который выводит найденные простые числа, должен выполняться, пока `k` меньше `5`.

Теперь функция `main` должна соответствовать коду, показанному в листинге Г.4.

Листинг Г.4. Функция main в файле calculate_primes_five_pthreads.cpp

```
...

int main() {
  int start = 0, end = 1000000;
  printf("Prime numbers between %d and %d:\n", start, end);

  std::chrono::high_resolution_clock::time_point duration_start =
      std::chrono::high_resolution_clock::now();

  pthread_t thread_ids[5];
  struct thread_args args[5];

  int args_start = 0;

  for (int i = 0; i < 5; i++) {
    args[i].start = args_start;
    args[i].end = (args_start + 199999);

    if (pthread_create(&thread_ids[i], NULL, thread_func, &args[i])) {
      perror("Thread create failed");
      return 1;
    }

    args_start += 200000;
  }

  for (int j = 0; j < 5; j++) {
    pthread_join(thread_ids[j], NULL);
  }

  std::chrono::high_resolution_clock::time_point duration_end =
      std::chrono::high_resolution_clock::now();

  std::chrono::duration<double, std::milli> duration =
      (duration_end - duration_start);

  printf("FindPrimes took %f milliseconds to execute\n", duration.count());

  printf("The values found:\n");
  for (int k = 0; k < 5; k++) {
    for(int n : args[k].primes_found) {
      printf("%d ", n);
    }
  }
  printf("\n");

  return 0;
}
```

Устанавливается значение 0

Устанавливается значение 5

Диапазон первого потока начинается с 0

Цикл повторяется, пока меньше 5

Конец диапазона теперь находится на 199 999 дальше, чем значение args_start

Увеличивается на 200 000

Устанавливается значение 5

Устанавливается значение, равное 5

Следующий шаг — компиляция файла `calculate_primes_five_pthreads.cpp`, а также выдача Emscripten команды сгенерировать файл шаблона HTML.

Откройте командную строку, перейдите в папку Appendix D\D.7.2\source\ и выполните следующую команду:

```
emcc calculate_primes_five_pthreads.cpp -O1 -std=c++11
➥ -s USE_PTHREADS=1 -s PTHREAD_POOL_SIZE=5
➥ -o five_pthreads.html
```

Выводы

В табл. Г.1 представлены результаты выполнения вычислений с использованием разного числа потоков. Каждый тест проводился по десять раз, а их продолжительность усреднялась:

- четыре потока pthread и вычисления также выполняются в основном потоке (см. главу 9);
- три потока pthread и вычисления также выполняются в основном потоке (см. подраздел«Решение для трех потоков» на с. 457);
- пять потоков без вычислений в основном потоке.

Таблица Г.1. Результаты выполнения вычислений с использованием разного числа потоков

Количество потоков	Firefox (мс)	Chrome (мс)
4 потока + основной	57,4	40,87
3 потока + основной	61,7	42,11
5 потоков (без вычислений в основном потоке)	52,2	36,06

Г.8. ГЛАВА 10

В главе 10 есть три упражнения.

Г.8.1. Упражнение 1

Какую функцию Node.js необходимо вызвать для загрузки файла JavaScript, созданного Emscripten?

Решение

`require`

Г.8.2. Упражнение 2

Какое свойство объекта `Module` в Emscripten нужно реализовать, чтобы получать информацию о том, когда модуль WebAssembly готов к взаимодействию?

Решение

```
onRuntimeInitialized
```

Г.8.3. Упражнение 3

Как бы вы изменили файл index.js из главы 8, чтобы логика динамического связывания работала в Node.js?

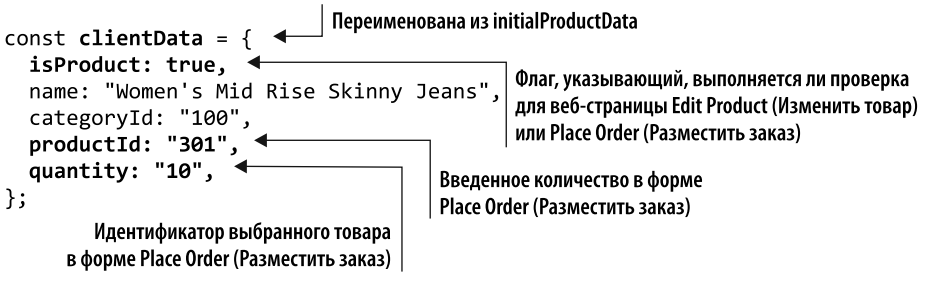
Решение

В папке WebAssembly\ создайте папку Appendix D\D.8.3\backend\, а затем выполните следующие шаги:

- скопируйте все файлы, кроме `index.html`, из папки Chapter 8\8.1 EmDynamicLibraries\frontend\ в только что созданную папку backend\;
- откройте файл `index.js` в своем любимом редакторе.

Файл `index.js` может вызываться веб-страницей Edit Product (Изменить товар) или Place Order (Разместить заказ), поэтому необходимо изменить объект `initialProductData`, чтобы он имел логический флаг (`isProduct`), указывающий, какие данные формы необходимо проверить. Вдобавок потребуется добавить два новых свойства для значений формы Place Order (Разместить заказ) (`productId` и `quantity`). Название самого объекта нужно будет изменить, чтобы лучше отразить его назначение.

Измените `initialProductData` в файле `index.js` в соответствии с кодом, показанным ниже:



Поскольку код на стороне сервера будет вызываться для проверки только одной веб-страницы за раз, вам не нужны глобальные переменные `productModule` и `orderModule`. Переименуйте переменную `productModule` в `validationModule`, а затем удалите строку кода `orderModule`. Выполните поиск кода и измените все экземпляры `productModule` и `orderModule`, чтобы использовать `validationModule`.

Следующий шаг — загрузка файла JavaScript, созданного Emscripten (`validate_core.js`). Для этого добавьте вызов функции `require`, показанный ниже, перед функцией `initializePage` в файле `index.js`:

```
const Module = require('./validate_core.js');
```

Модуль `validate_core` в WebAssembly был создан с помощью флага командной строки `MODULARIZE=1`. При использовании этого флага созданный Emscripten код JavaScript не запускается сразу после его загрузки. Код будет запущен только после того, как вы создадите экземпляр объекта `Module`. Поскольку код не запускается сразу после загрузки, вы не можете реализовать функцию `Module['onRuntimeInitialized']` в качестве отправной точки для кода.

Вместо этого заменим содержимое функции `initializePage` созданием экземпляра `validationModule` на основе того, что объект `clientData` указывает на необходимость проверки. При создании экземпляра объекта `Module` вы укажете функцию `onRuntimeInitialized` в этой точке.

Измените функцию `initializePage` в файле `index.js`, чтобы она соответствовала коду, показанному ниже:

```
function initializePage() {
  const moduleName = (clientData.isProduct ?
      'validate_product.wasm' : 'validate_order.wasm');

  validationModule = new Module({
    dynamicLibraries: [moduleName],
    onRuntimeInitialized: runtimeInitialized,
  });
}
```

Определяет, с каким файлом нужно связываться

Создает новый экземпляр Module, связывающийся с модулем с необходимой логикой проверки

Вызывает runtimeInitialized после загрузки модуля

После функции `initializePage` создайте функцию `runtimeInitialized`, которая будет вызывать функции `validateName` и `validateCategory`, в настоящее время находящиеся в функции `onClickSaveProduct`, если проверяются данные веб-страницы Edit Product (Изменить товар). В противном случае функция вызовет `validateProduct` и `validateQuantity`, в настоящее время находящиеся в функции `onClickAddToCart`, если проверяется веб-страница формы Place Order (Разместить заказ).

Добавьте код, показанный в листинге Г.5, в файл `index.js` после функции `initializePage`.

Листинг Г.5. Функция runtimeInitialized в файле index.js

```
...
function runtimeInitialized() {
  if (clientData.isProduct) {
    if (validateName(clientData.name) &&
        validateCategory(clientData.categoryId)) {
    }
  }
  else {
    if (validateProduct(clientData.productId) &&
      validateQuantity(clientData.quantity)) {
    }
  }
}
...
```

Необходимо проверить данные веб-страницы Edit Product (Изменить товар)

Проблем не обнаружено. Данные можно сохранить

Необходимо проверить данные веб-страницы Place Order (Разместить заказ)

Проблем не обнаружено. Данные можно сохранить

Следующий шаг — удаление из файла `index.js` функций пользовательского интерфейса:

- `switchForm`;
- `setActiveNavLink`;
- `setFormTitle`;
- `showElement`;
- `getSelectedDropdownId`;
- `onClickSaveProduct`;
- `onClickAddToCart`.

Когда в главе 8 создавался файл JavaScript, сгенерированный Emscripten, в него была включена функция `UpdateHostAboutError`, которая считывает сообщение об ошибке из памяти модуля, а затем вызывает функцию `setErrorMessage` в этом файле. Поскольку функция `UpdateHostAboutError` является частью JavaScript, загружаемого вызовом функции `require`, то ее область действия не позволяет ему получить доступ к функции `setErrorMessage` в этом файле. Чтобы функция `UpdateHostAboutError` имела доступ к функции `setErrorMessage`, необходимо изменить функцию `setErrorMessage` так, чтобы она стала частью глобального объекта. Кроме того, нужно изменить содержимое файла, чтобы использовать `console.log` для вывода сообщения об ошибке.

Обновите функцию `setErrorMessage` в файле `index.js`, чтобы она соответствовала коду, показанному ниже:

```
global.setErrorMessage = function(error) { console.log(error); }
```

Последнее изменение в файле `index.js` — добавление вызова функции `initializePage` в конце файла для запуска логики проверки. Добавьте следующий фрагмент кода в конец файла `index.js`:

```
initializePage();
```

Просмотр результатов

На данный момент в вашем объекте `clientData` содержатся только правильные данные, поэтому запуск кода прямо сейчас не покажет никаких ошибок валидации. Можно протестировать логику проверки количества, например изменив флаг `isProduct` на `false` и установив `quantity`, равное `0` (ноль).

Чтобы запустить файл JavaScript в Node.js, откройте командную строку, перейдите к Appendix D\D.8.3\backend\folder, а затем выполните следующую команду:

```
node index.js
```

Должно появиться сообщение о проверке: `Please enter a valid quantity`.

Г.9. ГЛАВА 11

В главе 11 есть два упражнения.

Г.9.1. Упражнение 1

При использовании WebAssembly Binary Toolkit для создания модуля WebAssembly какие узлы s-выражения должны появляться перед s-выражениями `table`, `memory`, `global` и `func`?

Решение

Если узлы `import` s-выражений добавлены, то они должны располагаться перед s-выражениями `table`, `memory`, `global` и `func`.

Г.9.2. Упражнение 2

Попробуйте изменить функцию `InitializeRowsAndColumns` в коде текстового формата так, чтобы теперь она поддерживала шесть уровней, а не три:

а) уровень 4 должен состоять из трех строк и четырех столбцов;

б) уровень 5 должен состоять из четырех строк и четырех столбцов;

в) уровень 6 должен состоять из четырех строк и пяти столбцов.

Решение

В папке WebAssembly\ создайте папку Appendix D\D.9.2\source\, а затем скопируйте файл `cards.wast` из папки Chapter 11\source\. Откройте файл `cards.wast`.

В функции `$InitializeRowsAndColumns` после третьего оператора `if` добавьте код, показанный в листинге Г.6.

Листинг Г.6. Дополнительный код для функции $InitializeRowsAndColumns

```
...

(func $InitializeRowsAndColumns (param $level i32)
  get_local $level
  i32.const 4
  i32.eq
  if
    i32.const 3
    set_global $rows

    i32.const 4
    set_global $columns
  end

  get_local $level
  i32.const 5
  i32.eq
  if
    i32.const 4
    set_global $rows

    i32.const 4
    set_global $columns
  end

  get_local $level
  i32.const 6
  i32.eq
  if
    i32.const 4
    set_global $rows

    i32.const 5
    set_global $columns
  end
)
...
```

Если запрошен уровень 4

Операторы if для уровней 1, 2 и 3 представлены здесь, но не показаны

Устанавливает значение rows, равное 3

Устанавливает значение columns, равное 4

Если запрошен уровень 5

Устанавливает значение rows, равное 4

Устанавливает значение columns, равное 4

Если запрошен уровень 6

Устанавливает значение rows, равное 4

Устанавливает значение columns, равное 5

Чтобы продолжить после уровня 3, необходима еще одна модификация. Измените глобальную переменную `$MAX_LEVEL`, чтобы теперь она содержала `i32.const 6`, как показано ниже:

```
(global $MAX_LEVEL i32 (i32.const 6))
```

Чтобы скомпилировать текстовый формат WebAssembly в модуль WebAssembly с помощью онлайн-инструмента wat2wasm, перейдите на сайт https://webassembly.github.io/wabt/demo/wat2wasm/. Замените текст на верхней левой панели инструмента содержимым вашего файла `cards.wast`, а затем скачайте модуль WebAssembly в папку Appendix D\D.9.2\source\folder. Назовите файл `cards.wasm`.

Создайте папку Appendix D\D.9.2\frontend\ и скопируйте в нее только что скачанный файл `cards.wasm`. Скопируйте все файлы, кроме `cards.wasm`, из папки Chapter 11\frontend\ в папку Appendix D\D.9.2\frontend\.

Чтобы просмотреть результаты, откройте браузер и введите `http://localhost:8080/game.html` в адресную строку, чтобы увидеть веб-страницу игры. Теперь игра должна позволить вам продолжить до уровня 6.

Г.10. ГЛАВА 12

В главе 12 есть два упражнения.

Г.10.1. Упражнение 1

Какими двумя способами можно получить доступ к переменной или вызвать функцию?

Решение

Можно получить доступ к переменной или вызвать функцию с помощью ее индекса, отсчитываемого от нуля. Кроме того, можно использовать имя элемента, если оно было указано.

Г.10.2. Упражнение 2

Возможно, вы заметили, что значение попыток не сбрасывается, когда вы переигрываете уровень или переходите на следующий. Используйте логирование в консоль браузера, чтобы определить источник проблемы.

Решение

В папке WebAssembly\ создайте папку Appendix D\D.10.2\source\, а затем скопируйте файл `cards.wast` из папки Chapter 12\source\. Откройте файл `cards.wast`.

Первое, что нужно сделать, — определить s-выражение импорта для функции ведения журнала `_Log`, которая принимает два параметра `i32`. Первый параметр — это указатель на ячейку памяти для строки, указывающей, из какой функции поступает значение журнала. Второй параметр — значение `$tries`.

JavaScript будет обрабатывать ведение журнала, поэтому функция `_Log`, показанная ниже, добавляется после импорта функции `_Pause`:

```
(import "env" "_Log" (func $Log (param i32 i32)))
```

Поиск кода всех функций, которые взаимодействуют со значением `$tries`, приводит к следующим результатам:

- `$InitializeCards`;
- `$PlayLevel`;
- `$SecondCardSelectedCallback`.

Узел `data` в конце файла `cards.wast` уже имеет имя функции для `SecondCardSelectedCallback`, поэтому нужно только добавить имена двух других функций. Добавьте символы `\0` (ноль и символ конца строки) между именами функций в качестве разделителя:

```
(data
  (i32.const 1024)
  "SecondCardSelectedCallback\0InitializeCards\0PlayLevel"
)
```

Вверху функции `$InitializeCards` после объявления локальной переменной `$count` поместите в стек значение `i32.const 1051`. Это начальная позиция узла `data` в памяти (`1024`) плюс количество символов, которые нужно получить до первого символа строки `InitializeCards` (`\0` — это один символ).

Добавьте в стек значение `$tries`, а затем вызовите функцию `$Log`:

```
i32.const 1051
get_global $tries
call $Log
```

В верхней части функции `$PlayLevel` повторите то, что вы сделали для функции `$InitializeCards`, но измените значение `i32.const` так, чтобы оно было в начале строки `PlayLevel`:

```
i32.const 1067
get_global $tries
call $Log
```

В верхней части функции `$SecondCardSelectedCallback` добавьте вызов `$Log`, передав `i32.const 1024` в качестве расположения строки в памяти:

```
i32.const 1024
get_global $tries
call $Log
```

Изменив текстовый формат, скомпилируйте текстовый формат WebAssembly в модуль WebAssembly с помощью онлайн-инструмента wat2wasm на сайте https://webassembly.github.io/wabt/demo/wat2wasm/. Замените текст на верхней левой панели инструмента содержимым вашего файла `cards.wast`, а затем скачайте WebAssembly в папку Appendix D\D.10.2\source\. Назовите файл `cards.wasm`.

Создайте папку Appendix D\D.10.2\frontend\ и скопируйте в нее только что скачанный файл `cards.wasm`. Скопируйте все файлы, кроме `cards.wasm`, из папки Chapter 12\frontend\ в папку Appendix D\D.10.2\frontend\, а затем откройте файл `game.js`.

Измените `sideImportObject` так, чтобы после функции `_Pause` следовала функция `_Log`, как показано ниже:

```
const sideImportObject = {
  env: {
    _Pause: pause,
    _Log: log,
  }
};
```

Остальные функции по-прежнему являются частью объекта, но не отображаются

В конце файла `game.js` добавьте следующую функцию `log`, которая считывает указанную строку из памяти и затем записывает информацию в окно консоли браузера:

```
function log(functionNamePointer, triesValue) {
  const name = getStringFromMemory(functionNamePointer);
  console.log(`Function name: ${name} triesValue: ${triesValue}`);
}
```

Если вы запустите файл `game.html` и откроете окно консоли инструментов разработчика браузера, то увидите, что вызовы функций логируются. Чтобы еще больше сузить радиус возможных причин проблемы, вы можете вызвать функцию `log` в любом другом месте кода.

В конце концов вы обнаружите, что источник проблемы находится в конце функции `$InitializeCards`. Значение глобальной переменной с индексом `6` помещается в стек, а затем глобальной переменной `$tries` присваивается значение, которое находится в стеке.

Если вы посмотрите на глобальные переменные, то обнаружите, что глобальная переменная `$tries` имеет индекс `6`. Вместо вызова `get_global 6` стеку должно быть присвоено значение `i32.const 0` для сброса переменной `$tries`, как показано в следующем фрагменте кода:

```
i32.const 0
set_global $tries
```

После того как проблема будет найдена, вызовы функции `$Log` можно удалить из файла `cards.wast`.

Г.11. ГЛАВА 13

В главе 13 есть два упражнения.

Г.11.1. Упражнение 1

Какую функцию Mocha вы бы использовали, если бы хотели сгруппировать несколько связанных тестов?

Решение

Функцию `describe`.

Г.11.2. Упражнение 2

Напишите тест, чтобы убедиться в том, что при передаче пустой строки для значения `categoryId` функции `ValidateCategory` возвращается правильное сообщение об ошибке.

Решение

В папке WebAssembly\ создайте папку Appendix D\D.11.2\tests\. Сделайте следующее:

- скопируйте файлы `validate.wasm`, `validate.js`, `package.json`, `tests.js` и `tests.html` из папки Chapter 13\13.2 tests\ в новую папку D.11.2\tests\;
- откройте командную строку и перейдите в папку D.11.2\tests\. Поскольку в файле `package.json` уже перечислены зависимости для Mocha и Chai, можно просто запустить следующую команду, и npm установит пакеты, перечисленные в файле:

  ```
  npm install
  ```

- откройте файл `tests.js` в своем любимом редакторе.

После теста `"Pass a string that's too long"` добавьте тест, который намеренно завершится ошибкой (листинг Г.7).

Листинг Г.7. Тестирование ValidateCategory с пустой строкой для categoryId

```
...

it("Pass an empty categoryId string to ValidateCategory", () => {
  const VALID_CATEGORY_IDS = [100, 101];
  const errorMessagePointer = Module._malloc(256);
  const categoryId = "";
  const expectedMessage = "something";

  const arrayLength = VALID_CATEGORY_IDS.length;
  const bytesPerElement = Module.HEAP32.BYTES_PER_ELEMENT;
  const arrayPointer = Module._malloc((arrayLength * bytesPerElement));
  Module.HEAP32.set(VALID_CATEGORY_IDS, (arrayPointer / bytesPerElement));

  const isValid = Module.ccall('ValidateCategory',
      'number',
      ['string', 'number', 'number', 'number'],
      [categoryId, arrayPointer, arrayLength, errorMessagePointer]);

  Module._free(arrayPointer);

  let errorMessage = "";
  if (isValid === 0) {
    errorMessage = Module.UTF8ToString(errorMessagePointer);
  }

  Module._free(errorMessagePointer);

  chai.expect(errorMessage).to.equal(expectedMessage);
});
```

Добавлен новый тест для проверки categoryId в функции ValidateCategory

Ожидаемое сообщение об ошибке; намеренно неправильно, поэтому тест не проходит

Проверяет, соответствует ли возвращенное сообщение ожидаемому

Чтобы запустить тесты, откройте командную строку, перейдите в папку D.11.2\tests\ и выполните следующую команду:

```
npm test tests.js
```

Новый тест должен провалиться.

Измените тест так, чтобы переменная `expectedMessage` теперь содержала значение `"A Product Category must be selected."` Если вы снова запустите тесты, они все должны пройти успешно.

Д

Дополнительные возможности текстового формата

В этом приложении

- ✓ Работа с операторами `if`.
- ✓ Работа с циклами.
- ✓ Раздел «Таблица» модуля WebAssembly и указатели на функции.

Как упоминалось в главе 11, выполнение кода в WebAssembly определяется в терминах стековой машины, в которой инструкции добавляют определенное количество значений в стек или извлекают их из него.

Когда функция вызывается впервые, стек для нее пуст. Платформа WebAssembly проверит его, когда функция завершится, чтобы гарантировать, что если она возвращает значение `i32`, то, например, последний элемент в стеке, когда функция его возвращает, должен быть значением `i32`. Если функция ничего не возвращает, то стек должен быть пуст на момент возврата функции. При наличии в стеке значения вы можете удалить этот элемент с помощью инструкции `drop`, которая вытолкнет верхний элемент из стека, как в следующем примере:

```
i32.const 1  ◄——— Добавляет значение 1 в стек
i32.const 2  ◄——— Добавляет значение 2 в стек
drop  ◄——— Извлекает значение 2 из стека
drop  ◄——— Извлекает значение 1 из стека
```

Бывают случаи, когда вам нужно выйти из функции до того, как она завершится. Для этого используется инструкция `return`, которая извлечет необходимые элементы из стека, а затем выйдет из функции. В следующем примере из стека будут извлечены два элемента, если они были единственными в стеке, а функция вернет `void`:

```
i32.const 1
i32.const 2
return
```

Если функция возвращает void, то инструкция return в этом случае извлечет два значения из стека

Д.1. ОПЕРАТОРЫ ПОТОКА УПРАВЛЕНИЯ

WebAssembly имеет несколько доступных операторов потока управления, таких как `block`, `loop` и `if`. Блоки и циклы не влияют на значения в стеке и представляют собой просто конструкции, которые имеют последовательность инструкций и метку. Блок может применяться в целях указания метки для использования с нужным в коде шаблоном ветвления.

Д.1.1. Операторы if

Написание блоков `if` интересно тем, что существует несколько способов их структурирования. Обе ветви `then` и `else` блока `if` необязательны. При использовании стиля стековой машины подразумевается оператор `then`. В обоих стилях — стека или вложенных s-выражений — вы можете использовать оператор `block` вместо оператора `then`, поскольку `block` — это просто серия инструкций с меткой.

Операторы `if` извлекают из стека значение `i32`, чтобы выполнить его проверку. Значение `0` (ноль) считается ложным, а любое ненулевое значение считается истинным. Поскольку оператору `if` необходимо извлечь значение `i32` из стека в стиле стековой машины, выполняется проверка, например `i32.eq`, перед оператором `if`, чтобы поместить логическое значение в стек. Стиль вложенного s-выражения может выполнять проверку либо до оператора `if`, либо внутри него.

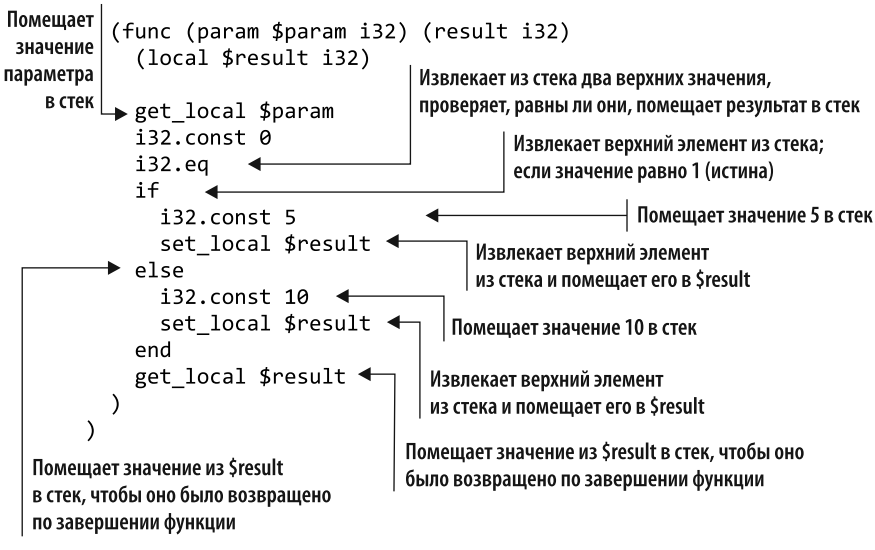
Посмотрим на оператор `if` в стиле стековой машины.

Оператор if в стиле стековой машины

Пример в листинге Д.1 представляет собой модуль, содержащий функцию, которая использует стиль стековой машины для проверки, равно ли значение параметра `0` (нулю). Если да, то функция вернет значение `5`. В противном случае вернет `10`.

Листинг Д.1. Пример блока if/else, написанного с использованием стиля стековой машины

```
(module
  (type $type0 (func (param i32) (result i32)))
  (export "Test" (func 0))

(func (param $param i32) (result i32)
  (local $result i32)

  get_local $param
  i32.const 0
  i32.eq
  if
    i32.const 5
    set_local $result
  else
    i32.const 10
    set_local $result
  end
  get_local $result
 )
)
```



Вы можете протестировать код, представленный в этом листинге, с помощью онлайн-инструмента wat2wasm.

Тестирование кода

Чтобы протестировать код, перейдите на сайт https://webassembly.github.io/wabt/demo/wat2wasm/ и скопируйте содержимое листинга Д.1 на верхнюю левую панель инструмента.

Как показано на рис. Д.1, на нижней левой панели инструмента вы можете заменить содержимое следующим фрагментом кода, чтобы загрузить модуль и вызвать функцию `Test`, передав значение `4`. Результат вызова функции `Test` будет отображаться в правом нижнем углу.

```
const wasmInstance = new WebAssembly.Instance(wasmModule, {});
console.log(wasmInstance.exports.Test(4));
```

Вы можете изменить значение, передаваемое в функцию `Test`, чтобы убедиться, что передача `0` (нуля) действительно возвращает `5`, а все остальные значения возвращают `10`. Посмотрим на вложенную версию s-выражения оператора `if`, которую мы только что видели в листинге Д.1.

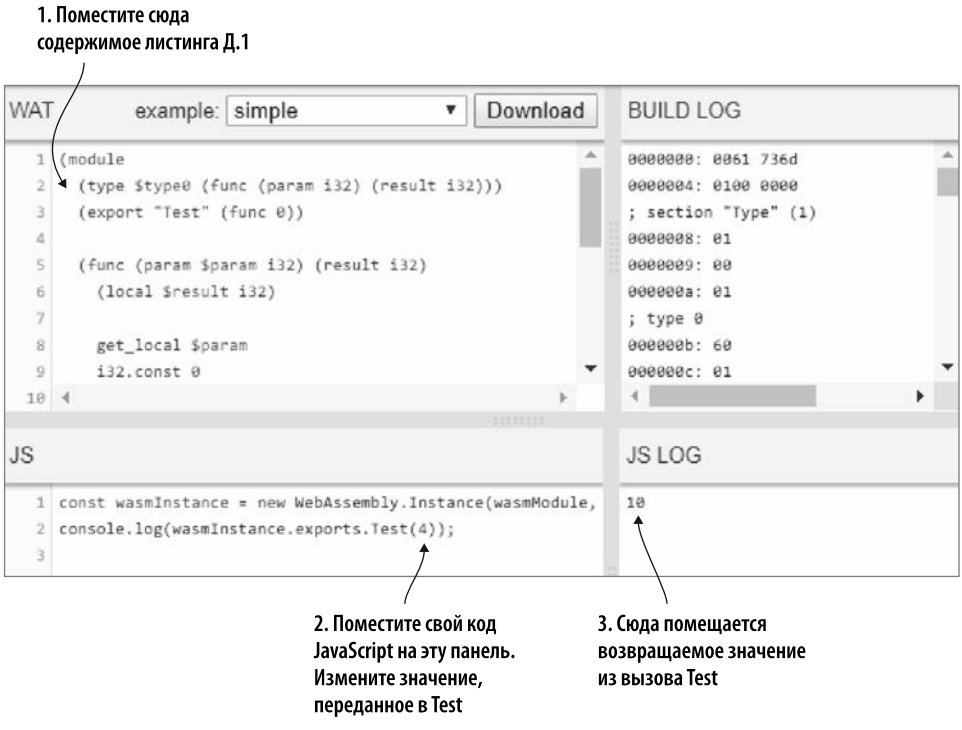


Рис. Д.1. Код из листинга Д.1 помещается на левую верхнюю панель, а код JavaScript — в левую нижнюю. Результат вызова функции отображается в правом нижнем углу

Оператор if в стиле вложенного s-выражения: проверка равенства перед оператором if

При использовании стиля стековой машины проверка равенства должна происходить перед оператором `if`, поскольку логическое значение должно уже находиться в стеке для данного оператора. В случае применения стиля вложенного s-выражения можно разместить проверку равенства перед оператором `if` или внутри него. В листинге Д.2 показан тот же код, что и в листинге Д.1, но вместо этого задействован стиль вложенного s-выражения.

Листинг Д.2. Стиль вложенного s-выражения с проверкой равенства перед оператором if

```
...

(func (param $param i32) (result i32)
  (local $result i32)

  (i32.eq                    ← Проверяет, равно ли значение параметра 0
    (get_local $param)
```

```
    (i32.const 0)
  )
  (if                                  Если проверка i32.eq была равна 1 (истина)...
    (then
      (set_local $result               ...устанавливает возвращаемое значение, равное 5
        (i32.const 5)
      )
    )                                  Проверка оператора if была равна 0 (ложь)...
    (else
      (set_local $result               ...устанавливает возвращаемое значение, равное 10
        (i32.const 10)
      )
    )
  )                                    Помещает возвращаемое значение в стек —
                                       оно будет возвращено по завершении функции
  (get_local $result)
)
...
```

Вы можете протестировать этот код, изменив содержимое верхней левой панели в онлайн-инструменте wat2wasm. JavaScript, который вы использовали на нижней панели для листинга Д.1, будет работать и для этого примера кода.

Посмотрим на пример, в котором проверка на равенство выполняется внутри оператора `if`.

Оператор if в стиле вложенного s-выражения: проверка равенства внутри оператора if

Хотя структура оператора `if` в листинге Д.2 имеет смысл, с учетом того, как работают проверки `if`, написание операторов `if` подобным образом обычно означает, что вы пишете их не так, как думают разработчики. При использовании стиля вложенного s-выражения можно изменить оператор `if`, чтобы проверка находилась в блоке оператора `if`, как показано в листинге Д.3.

Листинг Д.3. Пример, в котором проверка значения находится в блоке if

```
...

(func (param $param i32) (result i32)
  (local $result i32)

  (if
    (i32.eq                            Проверка равенства теперь
      (get_local $param)               находится внутри блока if
      (i32.const 0)
    )
    (then
      (set_local $result
        (i32.const 5)
```

```
      )
    )
    (else
      (set_local $result
        (i32.const 10)
      )
    )
  )

  (get_local $result)
)
...
```

Вы можете протестировать этот код, изменив содержимое верхней левой панели в онлайн-инструменте wat2wasm. JavaScript, который вы использовали на нижней панели для листинга Д.1, будет работать и для этого примера кода.

Операторы `if` могут использовать оператор `block` вместо оператора `then`.

Оператор if в стиле вложенного s-выражения: block вместо then

Если вы решите, что Emscripten должен выводить текстовый формат, эквивалентный двоичному файлу модуля, то заметите, что он использует операторы `block` вместо операторов `then`. Продемонстрируем вложенное s-выражение с оператором `if`, который применяет `block` вместо оператора `then`. Для этого изменим код в листинге Д.3, чтобы установить значение по умолчанию, равное `10`, для значения `$result` в начале функции. Присвоение переменной `$result` значения по умолчанию, равного `10`, позволяет удалить условие `else` из оператора `if`.

Измените оператор `if` так, чтобы он задействовал оператор `block` вместо оператора `then`, как показано в листинге Д.4.

Листинг Д.4. Пример условия if с использованием оператора block вместо then

```
...

(func (param $param i32) (result i32)
  (local $result i32)
  (set_local $result   <-- Присваивает значение по умолчанию, равное 10
    (i32.const 10)
  )

  (if
    (i32.eq
      (get_local $param)
      (i32.const 0)
    )
    (block   <-- Оператор then заменяется оператором block
      (set_local $result
```

```
        (i32.const 5)
      )
    )
  )

  (get_local $result)
)
...
```

В стиле стековой машины оператора `if` также может использоваться оператор `block` вместо оператора `then`.

Оператор if в стиле стековой машины: block вместо then

Вы можете изменить код из листинга Д.4, чтобы установить для переменной `$result` значение по умолчанию, равное `10`, в начале функции; это позволит удалить условие `else` из оператора `if`. Затем внутри оператора `if` заключите строки кода `i32.const` и `set_local` в оператор `block` и `end`, как показано в листинге Д.5.

Листинг Д.5. Предыдущий код в стиле стековой машины

```
...

(func (param $param i32) (result i32)
  (local $result i32)

  i32.const 10
  set_local $result  <-- Присваивает значение по умолчанию, равное 10

  get_local $param
  i32.const 0
  i32.eq  <-- Проверяет, равно ли значение параметра 0
  if
    block
      i32.const 5
      set_local $result
    end
  end

  get_local $result
)
...
```

Следующие операторы потока управления, о которых вы узнаете, — это циклы.

Д.1.2. Циклы

Для кода WebAssembly доступны три типа ветвления:

- `br` — переход к указанной метке;
- `br_if` — условный переход к указанной метке;
- `br_table` — таблица переходов для перехода к указанной метке.

Можно выполнить переход только к метке, определяемой конструкцией, внутри которой находится ветвь; это значит, например, что вы не можете перейти к середине цикла, когда ветвь находится вне его.

В цикле переходы к блоку эффективно действуют как оператор `break` в языках высокого уровня, тогда как переход к циклу действует как оператор `continue`. Цикл — это просто тип блока, который используется для формирования циклов.

Для того чтобы продемонстрировать, как работают циклы, создадим функцию `GetStringLength`, которая получает параметр `i32`, указывающий, где в памяти модуля находится строка, которую необходимо проверить. Функция вернет значение `i32` для длины строки.

Сначала создадим функцию, используя метод перехода к блоку (*действует как оператор* `break`), а затем, в следующем разделе, изменим цикл, чтобы вместо этого перейти к циклу (*действует как оператор* `continue`).

Оператор цикла вложенного s-выражения: переход от ветви к блоку

Прежде чем создавать функцию, необходимо определить память, которая будет использоваться модулем. Память определяется с помощью s-выражения с меткой `memory`, за которой следует необязательное имя переменной, начальное количество желаемых страниц памяти и, необязательно, максимальное количество желаемых страниц памяти. Каждая страница памяти составляет 64 Кбайт (65 536 байт).

Для данного модуля одной страницы памяти более чем достаточно, поэтому s-выражение памяти показано ниже:

```
(memory 1)
```

После того как этот модуль будет добавлен, создадим код JavaScript для онлайн-инструмента wat2wasm, который поместит строку в память модуля, а затем вызовет функцию `GetStringLength`. Поскольку JavaScript требует доступа к памяти модуля, необходимо его экспортировать. В следующем фрагменте кода показан оператор `export`, необходимый для памяти. Поскольку s-выражению `memory` не было присвоено имя переменной, указываем память по ее индексу:

```
(export "memory" (memory 0))
```

Функции `GetStringLength` нужны две локальные переменные: одна для отслеживания количества символов в строке на данный момент (`$count`), а вторая — для

отслеживания того, где функция читает в памяти в настоящее время (`$position`). Когда функция запускается, для `$count` будет установлено значение по умолчанию, равное 0, а для `$position` — значение полученного параметра: начальная позиция строки в памяти модуля.

Оператор `block` будет окружать цикл, из которого вы выйдете, если считанный из памяти символ будет символом конца строки. Оператору `block` будет дано имя переменной `$parent`. Внутри оператора `block` у вас будет оператор цикла с именем переменной `$while`.

В начале цикла текущий символ загружается из памяти, используя значения `$position`, с помощью инструкции `i32.load8_s`. Значение, загружаемое инструкцией `i32.load8_s`, — десятичная версия символа.

Затем инструкция `i32.eqz` проверит значение памяти, чтобы убедиться, что оно равно нулю (символ конца строки; символ «ноль» ASCII — это десятичное 48). Если значение равно нулю, то инструкция `br_if` переходит к блоку (`$parent`), таким образом выходит из цикла, и код продолжает выполнение после завершения цикла.

Если цикл не завершается, то переменные `$count` и `$position` увеличиваются на 1, а затем оператор `br` переходит в цикл, чтобы повторить его. После завершения цикла значение `$count` помещается в стек и возвращается вызывающей функции.

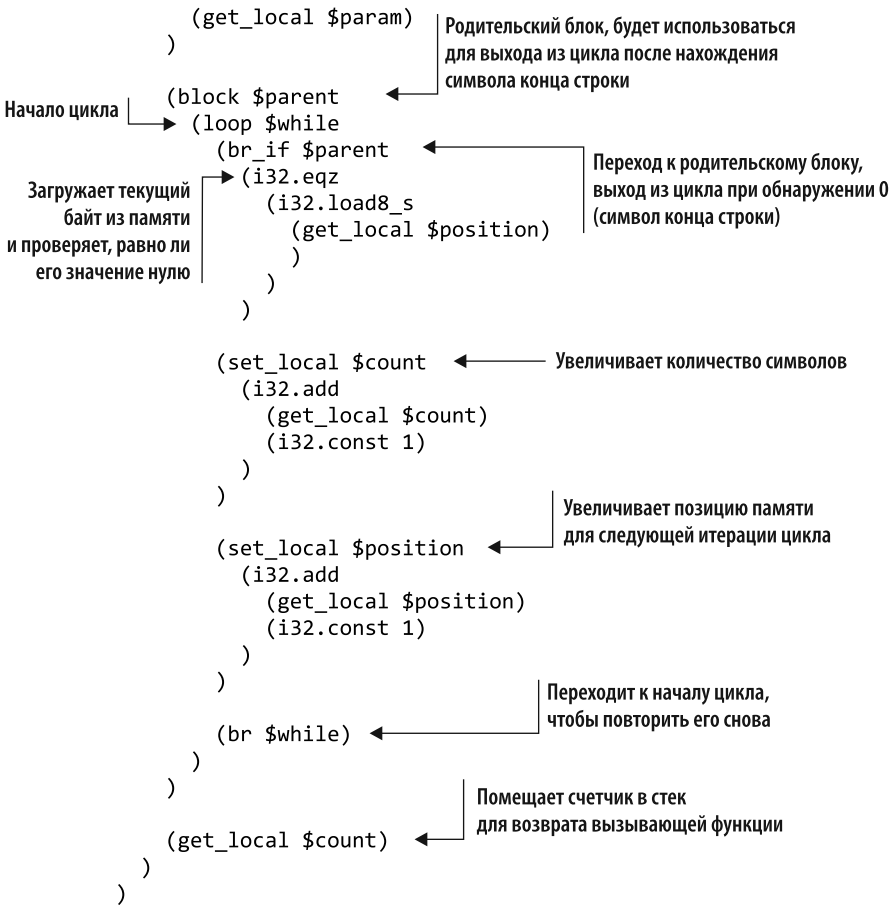
В листинге Д.6 представлен модуль, содержащий функцию `GetStringLength`.

Листинг Д.6. GetStringLength с использованием вложенных s-выражений и выходом из цикла

```
(module
  (type $type0 (func (param i32) (result i32)))

  (memory 1)

  (export "memory" (memory 0))
  (export "GetStringLength" (func 0))

  (func (param $param i32) (result i32)
    (local $count i32)
    (local $position i32)

    (set_local $count
      (i32.const 0)
    )

    (set_local $position
```

Будет хранить количество символов в строке, чтобы вернуть вызывающей функции

Текущая позиция в памяти модуля, которую нужно прочитать



Вы можете протестировать код, показанный в листинге Д.6, с помощью онлайн-инструмента wat2wasm.

Тестирование кода

Чтобы протестировать код, скопируйте содержимое листинга Д.6 на верхнюю левую панель онлайн-инструмента wat2wasm. На нижней левой панели (рис. Д.2) замените содержимое фрагментом кода, показанного ниже. Он загрузит модуль и поместит ссылку на память модуля в переменную `wasmMemory`. Определена функция `copyStringToMemory`, которая принимает строку и смещение памяти и записывает строку вместе с символом конца строки в память модуля.

Код вызывает функцию `copyStringToMemory`, передавая ей строку. Затем вызывается функция `GetStringLength` модуля, указывающая позицию в памяти, где

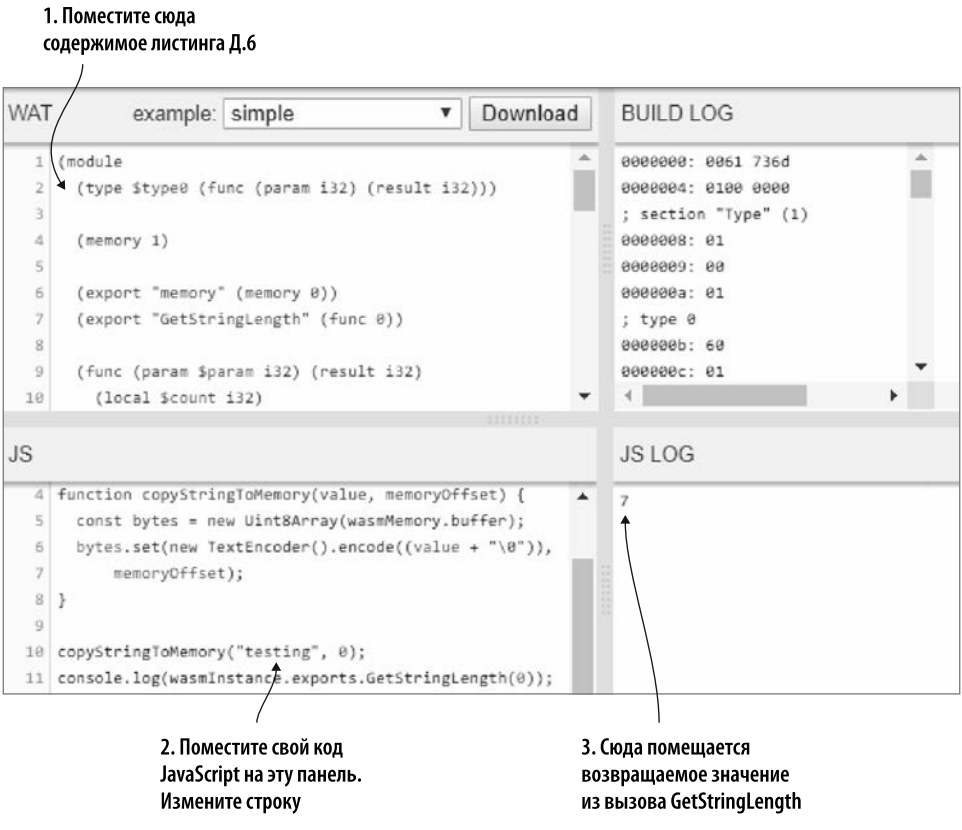


Рис. Д.2. Код из листинга Д.6 помещается на левую верхнюю панель, а код JavaScript — на левую нижнюю. Результат вызова функции отображается в правом нижнем углу

была записана строка. Результат вызова функции `GetStringLength` отображается на правой нижней панели:

```
const wasmInstance = new WebAssembly.Instance(wasmModule, {});
const wasmMemory = wasmInstance.exports.memory;

function copyStringToMemory(value, memoryOffset) {
  const bytes = new Uint8Array(wasmMemory.buffer);
  bytes.set(new TextEncoder().encode((value + "\0")),
      memoryOffset);
}

copyStringToMemory("testing", 0);
console.log(wasmInstance.exports.GetStringLength(0));
```

Вы можете изменить строку, передаваемую в функцию `copyStringToMemory`, чтобы проверить и увидеть, насколько разной может быть длина строки.

Посмотрим на версию только что созданного цикла для стековой машины.

Оператор if в стиле стековой машины: переход от ветви к блоку

Код, показанный в листинге Д.7, демонстрирует ту же функцию, которая была показана в листинге Д.6, но написан в стиле стековой машины.

Листинг Д.7. GetStringLength в стиле стековой машины и с выходом из цикла

```
...

(func (param $param i32) (result i32)
  (local $count i32)
  (local $position i32)

  i32.const 0
  set_local $count

  get_local $param
  set_local $position

  block $parent
    loop $while
      get_local $position
      i32.load8_s
      i32.eqz
      br_if $parent

      get_local $count
      i32.const 1
      i32.add
      set_local $count

      get_local $position
      i32.const 1
      i32.add
      set_local $position

      br $while
    end
  end

  get_local $count
)
...
```

Будет хранить количество символов в строке

Текущая позиция в памяти модуля, которую нужно прочитать

Загружает текущий байт из памяти и помещает его в стек

Значение равно нулю?

Если да, значит, вы достигли конца строки. Перейдите к родительскому блоку, чтобы выйти из цикла

Увеличивает значение $count

Увеличивает значение $position

Переход к началу цикла (цикл повторяется снова)

Помещает счетчик в стек для возврата вызывающей функции

Теперь изменим цикл, чтобы перейти от ветви к циклу вместо блока, который работает как оператор `continue`.

Оператор цикла вложенного s-выражения: переход от ветви к циклу

Работа с этой техникой требует изменить логику цикла, и подход «переход к циклу» не имеет окружающего оператора `block`. Если код не переходит в начало цикла, то цикл завершается. Новый цикл будет продолжаться, пока текущий символ не будет равен символу конца строки.

Измените код листинга Д.6, чтобы в нем больше не было s-выражения `block` вокруг s-выражения `loop`. Замените оператор `(br_if $parent` на `(if` для оператора `if`, а не оператора `branch`. Удалите закрывающую скобку из оператора `br_if`, который находится непосредственно перед строкой кода `(set_local $count`. Поместите закрывающую скобку для оператора `if` после оператора `(br $while)`.

Оператор `if` проверяет, не равен ли текущий символ нулю. Измените оператор `i32.eqz` (равный нулю) на оператор `i32.ne` (не равный нулю), а затем поместите следующее s-выражение после s-выражения `i32_load8`:

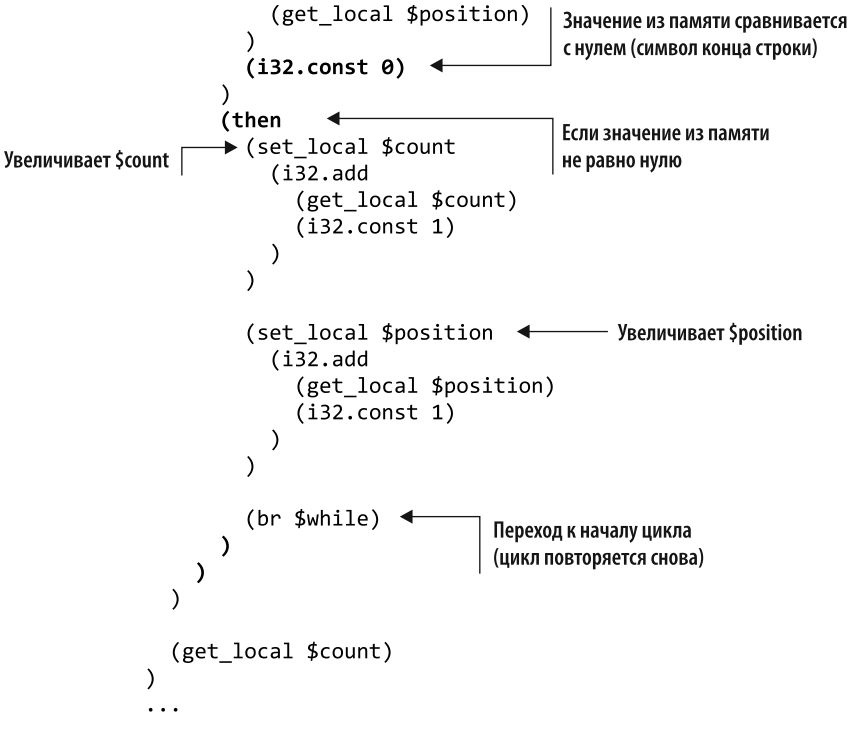
```
(i32.const 0)
```

После закрывающей скобки для s-выражения `i32.ne` поместите s-выражение `(then` с закрывающей скобкой после оператора `(br $while)`.

В листинге Д.8 показан измененный цикл с применением подхода `continue`.

Листинг Д.8. GetStringLength с использованием вложенных s-выражений и продолжением цикла

```
...

(func (param $param i32) (result i32)
  (local $count i32)
  (local $position i32)

  (set_local $count
    (i32.const 0)
  )

  (set_local $position
    (get_local $param)
  )

  (loop $while                    ← Начало цикла
    (if                           ← Заменяет оператор br_if
      (i32.ne                     ← Заменяет i32.eqz
        (i32.load8_s
```



Посмотрим на версию созданного цикла для стековой машины.

Оператор if в стиле стековой машины: переход от ветви к циклу

В листинге Д.9 показан тот же код, что и в листинге Д.8, но в стиле стековой машины.

Листинг Д.9. Предыдущий код в стиле стековой машины

```
...

(func (param $param i32) (result i32)
  (local $count i32)
  (local $position i32)

  i32.const 0
  set_local $count

  get_local $param
  set_local $position

  loop $while
```

```
    get_local $position
    i32.load8_s

    i32.const 0   <-- Новая проверка i32.ne
    i32.ne        <-- Заменяет i32.eqz
    if            <-- Заменяет br_if $parent
      get_local $count
      i32.const 1
      i32.add
      set_local $count

      get_local $position
      i32.const 1
      i32.add
      set_local $position

      br $while
    end
  end

  get_local $count
)
...
```

Следующая область, о которой вы узнаете, — как использовать раздел «Таблица» модуля для указателей функций.

Д.2. УКАЗАТЕЛИ НА ФУНКЦИИ

Модули WebAssembly имеют необязательный известный раздел «Таблица» (Table). Он представляет собой типизированный массив ссылок, таких как функции, которые не могут храниться в памяти в виде необработанных байтов по соображениям безопасности. Если бы адреса хранились в памяти модуля, то была бы вероятность, что вредоносный модуль попытается изменить адрес для доступа к данным, к которым доступа у него быть не должно.

Когда код модуля хочет получить доступ к данным, указанным в разделе «Таблица», он запрашивает, чтобы фреймворк WebAssembly работал с элементом по определенному индексу в таблице. Затем фреймворк WebAssembly считывает адрес, хранящийся в этом индексе, и выполняет действие.

Раздел «Таблица» определяется s-выражением, которое начинается с метки, использующей слово `table`; далее следует начальный размер (не обязательно), за которым следует максимальный размер; и наконец, далее следует тип данных, которые будут храниться в таблице. В настоящее время это лишь функции, поэтому применяется термин `funcref`.

СПРАВКА

Спецификация WebAssembly была изменена, чтобы использовать слово funcref вместо anyfunc для типа элемента таблицы. В выходном файле .wast Emscripten задействует новое имя, а WebAssembly Binary Toolkit может принимать код текстового формата, который использует любое имя. На момент написания этой книги инструменты разработчика в браузерах все еще использовали слово anyfunc при показе модуля. Firefox позволяет задействовать любое слово при создании объекта WebAssembly.Table в вашем JavaScript, однако на данный момент другие браузеры допускают только старое имя. Прямо сейчас для производственного кода JavaScript рекомендуется продолжать использовать anyfunc.

Чтобы продемонстрировать использование раздела «Таблица», мы создадим модуль, который импортирует две функции. Он будет иметь встроенную функцию, которая принимает параметр `i32`, указывающий индекс функции в разделе таблицы для вызова.

Первое, что потребуется модулю, — это два s-выражения `import` для двух функций, как показано ниже:

```
(import "env" "Function1" (func $function1))
(import "env" "Function2" (func $function2))
```

Затем нужно определить s-выражение `table` размером `2` для двух функций:

```
(table 2 funcref)
```

После s-выражения `table` следует s-выражение `export` для функции, которую JavaScript вызовет, чтобы указать, какую функцию нужно вызвать далее:

```
(export "Test" (func $test))
```

При создании модуля необходимо, чтобы импортированные функции были добавлены в раздел «Таблица». Для этого нужно определить s-выражение `element`. Сущности в этом s-выражении будут добавлены в раздел «Таблица» автоматически при создании экземпляра модуля.

S-выражение `element` начинается с метки `elem`, за которой следует начальный индекс в таблице, куда будут помещены ссылки на объекты, а затем следуют элементы для размещения в разделе «Таблица». Следующий фрагмент кода добавит две функции в раздел «Таблица», начиная с индекса таблицы `0` (ноль):

```
(elem (i32.const 0) $function1 $function2)
```

Следующий шаг — определение вашей функции `$test`, которая получает значение параметра `i32` и не имеет возвращаемого значения, как показано ниже:

```
(func $test (param $index i32)
)
```

В функции `$test` нужно вызвать запрошенный элемент таблицы. Чтобы вызвать элемент в разделе «Таблица», вы передаете индекс в инструкцию `call_indirect` и вдобавок указываете тип (сигнатуру функции), который вызываете так, как показано ниже:

```
(call_indirect (type $FUNCSIG$v)
  (get_local $index)
)
```

$FUNCSIG $v — имя переменной для s-выражения типа (также можно использовать индекс)

Код модуля, получившегося в результате, показан в листинге Д.10.

Листинг Д.10. Модуль указателя на функцию, использующий стиль вложенного s-выражения

```
(module
  (type $FUNCSIG$v (func))

  (import "env" "Function1" (func $function1))
  (import "env" "Function2" (func $function2))

  (table 2 funcref)

  (export "Test" (func $test))

  (elem (i32.const 0) $function1 $function2)

  (func $test (param $index i32)
    (call_indirect (type $FUNCSIG$v)
      (get_local $index)
    )
  )
)
```

Сигнатуры двух функций, которые будут импортированы

Создает таблицу с начальным размером 2

Помещает две функции в таблицу, начиная с индекса 0

Вызывает элемент в таблице, используя индекс, полученный как параметр

Закончив изменять код модуля, можно его протестировать.

Д.2.1. Тестирование кода

Чтобы протестировать код, скопируйте содержимое листинга Д.10 на верхнюю левую панель онлайн-инструмента wat2wasm. На нижней левой панели (рис. Д.3) замените содержимое фрагментом кода, показанного ниже. Он будет определять объект `importObject` для модуля, содержащего две импортируемые функции.

Каждая функция будет записывать сообщение в консоль инструментов разработчика браузера, указывающее, какая функция была вызвана.

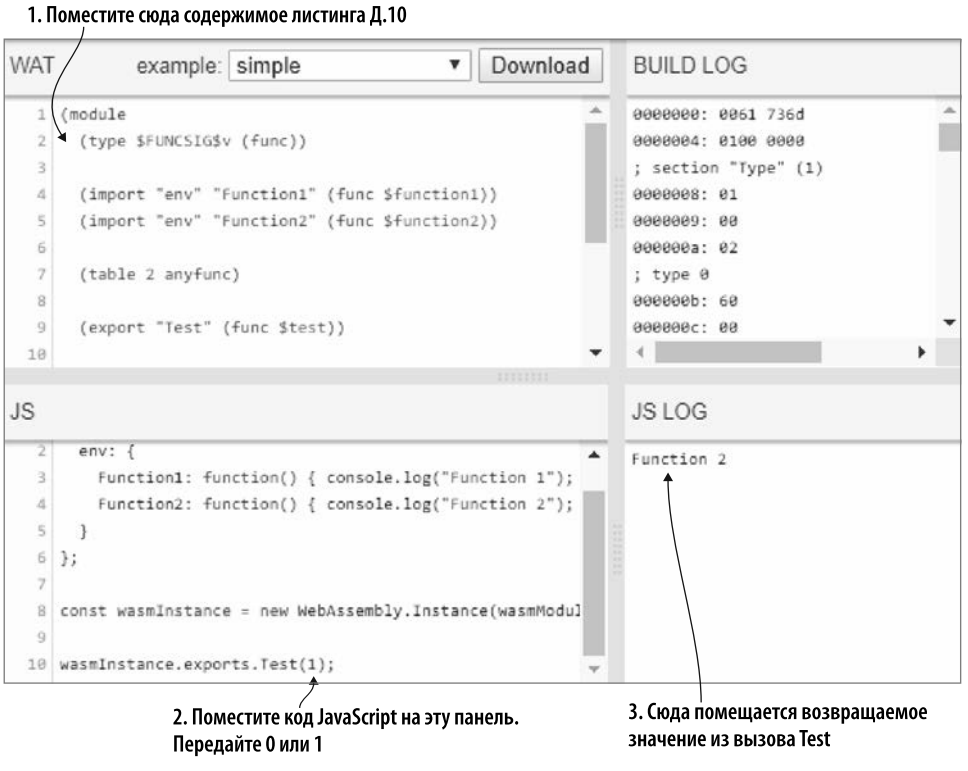
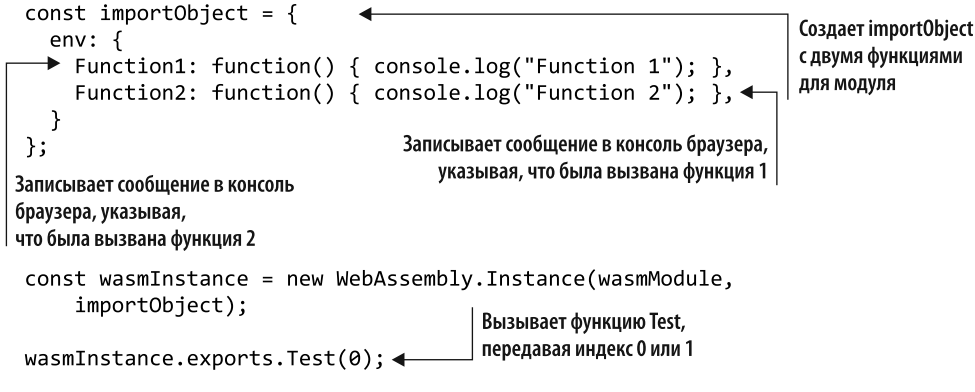


Рис. Д.3. Код из листинга Д.10 помещается на левую верхнюю панель, а код JavaScript — на левую нижнюю. Результат вызова функции отображается в правом нижнем углу

При наличии экземпляра модуля вы можете вызвать функцию `Test`, передав `0` или `1`, чтобы вызывать функции из раздела «Таблица»:



Жерар Галлан

WebAssembly в действии

Перевел с английского *А. Павлов*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Научный редактор	*Н. Зимин*
Литературный редактор	*Н. Хлебина*
Художественный редактор	*В. Мостипан*
Корректоры	*М. Молчанова, Е. Павлович*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2021. Наименование: книжная продукция. Срок годности: не ограничен.

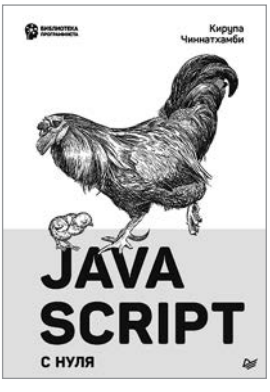
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 16.09.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 39,990. Тираж 700. Заказ 0000.

Кирупа Чиннатхамби

JAVASCRIPT С НУЛЯ



JavaScript еще никогда не был так прост! Вы узнаете все возможности языка программирования без общих фраз и неясных терминов. Подробные примеры, иллюстрации и схемы будут понятны даже новичку. Легкая подача информации и живой юмор автора превратят нудное заучивание в занимательную практику по написанию кода. Дойдя до последней главы, вы настолько прокачаете свои навыки, что сможете решить практически любую задачу, будь то простое перемещение элементов на странице или даже собственная браузерная игра.

Яков Файн, Антон Моисеев

TYPESCRIPT БЫСТРО

«TypeScript быстро» научит вас секретам продуктивной разработки веб- или самостоятельных приложений. Она написана практиками для практиков.

В книге разбираются актуальные для каждого программиста задачи, объясняется синтаксис языка и описывается разработка нескольких приложений, в том числе нетривиальных — так вы сможете понять, как использовать TypeScript с популярными библиотеками и фреймворками.

Вы разберетесь с превосходным инструментарием TypeScript и узнаете, как объединить в одном проекте TypeScript и JavaScript. Среди продвинутых тем, рассмотренных авторами, — декораторы, асинхронная обработка и динамические импорты.

Прочитав эту книгу, вы поймете, что именно делает TypeScript особенным.

Никола Лейси

PYTHON, НАПРИМЕР

Это Python, например! Познакомьтесь с самым быстрорастущим языком программирования на сегодняшний день.

Легкое и увлекательное руководство поможет шаг за шагом прокачать навыки разработки. Никаких архитектур компьютера, теорий программирования и прочей абракадабры — больше практики! В книге 150 задач, которые плавно перенесут читателя от изучения основ языка к решению более сложных вещей. Руководство подойдет всем, у кого голова идет кругом от технического жаргона и пространных объяснений — автор уверен, что учить можно и без этого.

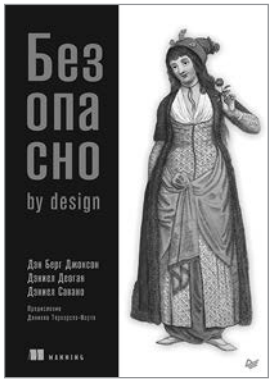
Роберт С. Сикорд

ЭФФЕКТИВНЫЙ С. ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Мир работает на коде, написанном на C, но в большинстве учебных заведений программированию учат на Python или Java. Книга «Эффективный C для профессионалов» восполняет этот пробел и предлагает современный взгляд на C. Здесь рассмотрен C17, а также потенциальные возможности C2x. Издание неизбежно станет классикой, с его помощью вы научитесь писать профессиональные и надежные программы на C, которые лягут в основу устойчивых систем и решат реальные задачи.

Дэн Берг Джонсон, Дэниел Деоган, Дэниел Савано

БЕЗОПАСНО BY DESIGN



«Безопасно by Design» не похожа на другие книги по безопасности. В ней нет дискуссий на такие классические темы, как переполнение буфера или слабые места в криптографических хэш-функциях. Вместо собственно безопасности она концентрируется на подходах к разработке ПО. Поначалу это может показаться немного странным, но вы поймете, что недостатки безопасности часто вызваны плохим дизайном. Значительного количества уязвимостей можно избежать, используя передовые методы проектирования. Изучение того, как дизайн программного обеспечения соотносится с безопасностью, является целью этой книги. Вы узнаете, почему дизайн важен для безопасности и как его использовать для создания безопасного программного обеспечения.

Мартин Одерски, Лекс Спун, Билл Веннерс

SCALA. ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ. 4-е издание

«Scala. Профессиональное программирование» — главная книга по Scala, популярному языку для платформы Java, в котором сочетаются концепции объектно-ориентированного и функционального программирования, благодаря чему он превращается в уникальное и мощное средство разработки.

Этот авторитетный труд, написанный создателями Scala, поможет вам пошагово изучить язык и идеи, лежащие в его основе.

Данное четвертое издание полностью обновлено. Добавлен материал об изменениях, появившихся в Scala 2.13, в том числе:

- новая иерархия типов коллекций;
- новые конкретные типы коллекций;
- новые методы, добавленные к коллекциям;
- новые способы определять собственные типы коллекций;
- новые упрощенные представления.