Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте *www.dmkpress.com*, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу *dmkpress@gmail.com*; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу *http://dmkpress.com/authors/publish_book/* или напишите в издательство по адресу *dmkpress@gmail.com*.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу *dmkpress@gmail.com*.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Вступительное слово Ольве Маудала

Впервые я встретил Андерса Шау Кнаттена на местной встрече программистов C++ в Осло. С тех пор прошло довольно много времени, мы подружились и вместе проводили время, сталкиваясь на разных конференциях. Андерс проводил презентации на таких посвященных C++ мероприятиях, как ACCU, NDC TechTown, CppCon, C++ on Sea, Meeting C++ и многих других. Посещая их, я всегда узнаю что-то новенькое. Андерс может часами говорить о том, как на самом деле ведут себя целые числа, о том, что происходит перед вызовом main(), или о том, как эффективно работать с отладчиком. Он также обладает удивительной способностью заметить что-то, переосмыслить, придать другую форму и вывести на следующий уровень. Именно это он и сделал, посетив два моих печально известных семинара C++ Pub Quiz. Очень скоро Андерс создал изумительный сайт cppquiz.org[1] (загляните, не поленитесь!). А теперь он проводит свои собственные, значительно улучшенные версии таких семинаров на конференциях. В этой книге Андерс продолжил развивать эту идею.

Я люблю книги, люблю C++ и люблю задачки. В книге «Хорошо ли вы знаете C++?» все это сошлось в одном месте. Я изучаю и пользуюсь C++ вот уже больше 30 лет, но уверен, что буду читать и перечитывать эту книгу снова и снова и каждый раз находить что-то новое. Если вы глубоко привязаны к C++, то тоже захотите прочитать ее. Книга станет для вас дорогой к более глубокому освоению этого чрезвычайно мощного и чарующего языка программирования. Она содержит 25 тщательно отобранных задач, которые призваны испытать, насколько хорошо вы понимаете современные версии языка. После того как вы попытаетесь решить задачу самостоятельно, Андерс представит глубокое и основательное объяснение причин, по

[1] *https://cppquiz.org*.

которым эта задача интересна, и способов научиться рассуждать о таких вещах.

Вы можете написать на C++ красивые программы, но, конечно, можно сочинить и полную лажу. Важно понимать, что такое C++, потому что он уходит корнями в языки BCPL, C, Simula 67 и Algol 68. C++ стал тем, чем он является, потому что был быстро принят сообществом и набрал миллионы активных пользователей уже на протяжении самых первых лет. C++ стал таким, каким мы его знаем, потому что он доверяет программисту и сосредоточен на решении реальных задач. Да, C++ сложен, быть может, даже слишком сложен, но это язык, на который нужно go-to (игра слов не случайна!) при работе во многих областях, как то: операционные системы, встраиваемые системы, высокопроизводительные вычисления, особо быстрые приложения, видеоигры и космические станции. Приведем высказывание Бьярна Страуструпа (создателя C++), которое все расставляет по местам:

> Есть только два вида языков: те, на которые жалуются, и те, которыми никто не пользуется.

Желаю вам всего наилучшего в путешествии к более глубокому пониманию C++. На этот раз вашим гидом будет Андерс. Получайте удовольствие!

Ольве Маудал,
докладчик на международных конференциях по C и C++,
преподаватель и организатор конференций.
Июнь 2024

Благодарности

Прежде всего хочу поблагодарить Ольве Маудала и Ассоциацию пользователей C и C++ (ACCU), которые побудили меня создать сайт CppQuiz.org[1] во время конференции ACCU 2013. Я также благодарен всем, кто помогал в этом предприятии, наполняя сайт материалами, участвуя в реализации и присылая извещения об ошибках.

Большое спасибо моему редактору, Сандре Уильямс, за бесценную помощь на протяжении всей работы. Без твоего руководства, поддержки и советов эта книга оказалась бы гораздо хуже. Я также признателен Маргарет Элдридж, которая обратилась ко мне от имени издательства, и Франсису Буонтемпо, который познакомил нас.

Мне посчастливилось получать помощь от лучших известных мне специалистов по C++. Спасибо вам, Даниэла Энгерт, Бьярн Фаллер, Ольве Маудал, Картхик Нишант, Том Шульц, Тина Ульбрих, Сергей Васильченко и Пётр Вирчиньский, за работу в качестве технических редакторов!

И наконец, я хочу поблагодарить свою жену, Шарлотту, чья поддержка позволила мне посвятить время и силы написанию этой книги, а также Кристиана и Себастьяна, которые все время подбадривали меня.

[1] *https://cppquiz.org*.

Предисловие

C++ – один из самых больших и старых языков программирования, которым активно пользуются. Он знаменит тем, что превратно истолковывает все поведения по умолчанию и, как в знаменитом в сообществе C++ примере, может заставить демонов вылетать у вас из носа[1]. Невозможно выбрать лучший язык для книги, посвященной головоломкам в программировании!

На примере 25 задач мы изучим, как C++ работает под капотом, и, в частности, уделим внимание нескольким его важным причудам. Чтобы извлечь максимум пользы из книги, вы должны иметь опыт работы на C++ и быть знакомы с основами языка, в том числе с простым объектно ориентированным программированием и шаблонами. Прочитав книгу, вы будете более глубоко разбираться в таких вопросах, как инициализация, время жизни, разрешение перегрузки, неявные преобразования, наследование, неопределенное и неспецифицированное поведение и др. Но, на мой взгляд, более важно то, что вы заинтересуетесь, как на самом деле работает C++, пусть даже в одной книге можно дать ответы лишь на немногие вопросы.

Как пользоваться этой книгой

Книга содержит 25 задач по C++ с ответами и объяснениями. Большая их часть представляет собой законченные программы, которые, согласно стандарту C++, дают вполне определенный результат. Но некоторые приводят к ошибкам компиляции и даже к неопределенному поведению. Ваша задача – понять, что произойдет после того, как вы откомпилируете и выполните каждую задачу, воспользовавшись отвечающей стандарту реализацией C++.

Возьмем для примера следующую гипотетическую задачу. Это полная программа на C++ с функцией `main`:

---

[1] *http://catb.org/jargon/html/N/nasal-demons.html*.

```
#include <iostream>
int main()
{
    std::cout << (1 < 2);
}
```

Ваша задача – прочитать код и попытаться угадать, что выведет откомпилированная программа. Обязательно подумайте сами, прежде чем переворачивать страницу в поисках ответа!

Хочу обратить внимание на несколько технических деталей.

- Для краткости я объявляю `main` без параметров (`argc`, `argv`) и не возвращаю явно значение. То и другое необязательно, а без них задачи немного проще читать.
- Во всех задачах я использую `struct`, а не `class`. Семантической разницы между ними нет – просто в `struct` по умолчанию подразумевается видимость членов `public`, а не `private`, поэтому нам не нужно вставлять всюду `public:`.
- Как всегда в C++, значения типа `bool` по умолчанию печатаются как `1` и `0`, а не `true` и `false`.

Эта программа печатает `1` (представляющую `true`), потому что `1` меньше `2`.

Но получить правильный ответ – только полдела. А вторая половина – понять, *почему* программа работает именно так, а не иначе. Задачи убеждают в том, что нужно лучше изучать, как C++ работает за кулисами. Я призываю вас вчитываться в объяснения, пока не достигнете полного понимания.

Неопределенное поведение

В некоторых задачах может иметь место неопределенное поведение. Так называется ситуация, когда при выполнении программы происходит что-то ужасное, чего компилятор не может (точнее говоря, не обязан) обнаружить. Например, мы можем обратиться к элементу за пределами массива, или арифметическое выражение, содержащее целые со знаком, может привести к переполнению. В таких случаях стандарт C++ не налагает никаких ограничений на реализацию, и случиться может все что угодно, включая демонов, вылетающих из носа. Если в задаче имеется неопределенное поведение, то вы должны не только идентифицировать его, но и предположить, что произойдет на практике в типичной системе. Действительно ли из носа

начнут вылетать демоны или случится что-то более приземленное и определенное?

Снова рассмотрим пример:

```
#include <iostream>
#include <limits>
int main()
{
    std::cout << std::numeric_limits<int>::max() + 1;
}
```

Переполнение в целочисленной арифметике со знаком – неопределенное поведение, поэтому, обнаружив его, вы на полпути к решению!

Строить догадки о том, что произойдет в случае неопределенного поведения, – занятие неблагодарное. Поэтому поступим иначе. В любой задаче с неопределенным поведением вторая половина – выяснить, что произойдет при запуске программы на своем компьютере. На моем компьютере целые со знаком представляются в дополнительном коде (как во всех реализациях, отвечающих стандарту C++20 и более поздним), и мой процессор не возбуждает исключения в случае переполнения. Поэтому, когда я прибавляю `1` к наибольшему положительному целому, происходит оборачивание и получается наименьшее отрицательное целое. Поскольку в моей системе тип `int` 32-разрядный, программа печатает `-2147483648`. Точное значение вам ни к чему, но если вы догадались, что будет напечатано наименьшее отрицательное целое, значит задача решена!

Не делайте этого дома

Угадывание или проверка того, что происходит в случае неопределенного поведения, – интересное упражнение, которое может пополнить ваши знания о работе C++ на своей платформе. Также это поможет распознавать определенные типы ошибок в реальных программах. Но не делайте никаких предположений о своих реальных программах на основе этих находок! Ваши предположения могут оказаться ложными на других компьютерах, после обновления компилятора или при компиляции с другими параметрами оптимизации. Компилятору даже разрешено удалять проверку ошибок из кода, если он может доказать, что имеет место неопределенное поведение!

Неспецифицированное и зависящее от реализации поведение

Стандарт C++ не все определяет строго, он оставляет некоторую свободу реализации. Вот несколько примеров:

- конкретные размеры целых типов;
- порядок вычисления аргументов функции;
- порядок инициализации глобальных переменных.

Это позволяет каждой реализации принимать решения, наиболее подходящие в конкретной системе.

В большинстве программ какое-то неспецифицированное или зависящее от реализации поведение присутствует, и это не ошибка. В отличие от неопределенного поведения демоны из носа не полетят. Просто разные реализации могут вести себя немного по-разному, не выходя за рамки допустимого поведения.

Если в задаче имеет место неспецифицированное или зависящее от реализации поведение, попытайтесь догадаться, как поведет себя программа в типичном случае.

Эксперименты с кодом

Самая важная часть изучения всего, связанного с программированием, – самостоятельные эксперименты. Код из этой книги можно найти на ее домашней странице на сайте The Pragmatic Bookshelf[1]. Вы можете собрать его локально, открыв файл `CMakeLists.txt` в своей любимой IDE или в командной строке:

```
mkdir build
cd build
cmake ..
cmake --build .
```

Проект содержит по одному `cpp`-файлу на задачу, и каждый из них транслируется в один двоичный файл, названный так же, как соответствующая задача в книге.

Можете также повозиться с кодом непосредственно в браузере, скопировав его в онлайновый компилятор. Я горячо рекомендую

[1] *https://pragprog.com/titles/akbrain*.

сайт Compiler Explorer[1], где вы можете выбрать разные компиляторы и сравнить версии и архитектуры, попробовать различные параметры оптимизации, добавить другие флаги компилятора, параметры проверки кода и т. д.

Итак, приступим! Да – и берегитесь демонов.

[1] *https://godbolt.org*.

Задача 1

Сколько градусов?

how-many-degrees.cpp

```
#include <iostream>

struct Degrees
{
    Degrees() : degrees_(0)
    {
        std::cout << "Default constructed\n";
    }
    Degrees(double degrees) : degrees_(degrees)
    {
        std::cout << "Constructed with " << degrees_ << "\n";
    }
    double degrees_;
};
struct Position
{
    Position() : latitude_{1} { longitude_ = Degrees{2}; }
    Degrees latitude_;
    Degrees longitude_;
};

int main()
{
    Position position;
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
Constructed with 1
Default constructed
Constructed with 2
```

Обсуждение

В структуре `Position` два члена типа `Degrees`. Один инициализируется в списке инициализации членов, второй в теле конструктора. Тогда почему мы видим *три* сконструированных объекта `Degrees`?

Все члены класс инициализируются до входа в тело конструктора. Если вы явно инициализируете член, как мы поступили с `latitude_`, то именно эта инициализация и будет использована. Если нет, как в случае с `longitude_`, то член будет инициализирован по умолчанию, поэтому мы видим, что напечатана строка `Default constructed`. После того как все члены инициализированы, мы входим в тело конструктора, инициализируем временный объект `Degrees{2}`, а затем присваиваем его копированием ранее инициализированному по умолчанию члену `longitude_`.

Заметим, что вместо явной инициализации члена в списке инициализации членов можно было бы использовать инициализатор по умолчанию прямо в объявлении члена:

examples/default-member-initializer/default-member-initializer.cpp

```
struct Position
{
    Position() { longitude = Degrees{2}; }
    Degrees latitude{1}; // инициализатор члена по умолчанию
    Degrees longitude;
};
```

Использование инициализатора члена по умолчанию может быть полезной идеей, если конструкторов несколько. Тогда не нужно помнить о включении этого члена в каждый список инициализации членов.

Для дополнительного чтения

Конструкторы и списки инициализации членов
https://en.cppreference.com/w/cpp/language/constructor.

Задача 2

Теория струн

string-theory.cpp

```
#include <iostream>
#include <string>

void serialize(const void*) { std::cout << "const void*"; }
void serialize(const std::string&) { std::cout << "const string&"; }
int main()
{
    serialize("hello world");
}
```

Угадайте результат

Попробуйте угадать результат, не переворачивая страницу.

Программа печатает следующий результат:

```
const void*
```

Обсуждение

Почему передача строки функции `serialize` приводит к вызову перегруженного варианта, который принимает указатель на `void`, а не к вызову варианта, принимающего строку?

Когда мы вызываем функцию, имеющую несколько перегруженных вариантов, компилятор применяет процесс *разрешения перегрузки*, чтобы понять, какой вариант самый подходящий. Для этого компилятор пытается преобразовать каждый аргумент функции к типу соответствующего параметра для каждого перегруженного варианта. Одни преобразования считаются лучше других, а самое лучшее – когда аргумент уже имеет правильный тип.

Перегруженные варианты, в которых все аргументы можно успешно преобразовать, помещаются в множество *подходящих функций*. Затем компилятор должен определить, какой вариант выбрать из этого множества. Если некоторый перегруженный вариант имеет лучшее преобразование, чем другие, хотя бы для одного аргумента и не худшее для всех остальных, то этот вариант считается *лучшей подходящей функцией* и является результатом процесса разрешения перегрузки. Если ни один вариант не лучше всех остальных, то вызов считается недопустимым и не компилируется.

Рассмотрим пример:

```
serialize(int, int); // 1
serialize(float, int); // 2
```

При таких двух перегруженных вариантах предположим, что имеется вызов:

```
serialize(1, 2);
```

Оба перегруженных варианта `serialize` являются подходящими. Но для первого преобразование первого аргумента лучше (`int` → `int` лучше, чем `int` → `float`), а для второго – не хуже (`int` → `int` в обоих вариантах), поэтому он выбирается как лучшая подходящая функция.