

MINI DOOM

Современный C++ через игру

```
game.cpp — как устроен мир изнутри
#include "game.hpp"

#include <string>
#include <array>
#include <cs_lib>
#include <iostream>
#include <string_view>

namespace doom {

namespace {

constexpr int kWidth = 12;
constexpr int kHeight = 5;
constexpr std::array<std::string_view, kHeight> kLevel{
    "#####",
    "#@..#.z.h.X#",
    "#.#.#.###..#",
    "#h..i.....#",
    "#####",
};

constexpr auto cellAt(Pos p) {
    const auto row = static_cast<std::size_t>(p.y);
    const auto col = static_cast<std::size_t>(p.x);
    return kLevel[row][col];
};

constexpr auto nameOf(Kind kind) {
    switch (kind) {
        case Kind::Imp: return "imp";
        case Kind::Zombie: return "zombi";
    }
    return "???";
};

} // namespace

Game(unsigned seed) : rng_(seed) {
    for (int y = 0; y < kHeight; ++y) {
        for (int x = 0; x < kWidth; ++x) {
            const Pos p{x, y};
            const auto cell = cellAt(p);
            if (cell == '@') {
                pos_ = p;
                break;
            }
            if (cell == 'X') {
                monsters_.push_back({Kind::Imp, p, 20});
            }
            if (cell == 'z') {
                monsters_.push_back({Kind::Zombie, p, 20});
            }
        }
    }
}
```



ТОМ 1

Учимся держать оружие

C++20 · Visual Studio 2026

18+ 8+

Алексей Примак

**MINI DOOM современный
C++ через игру том 1
учимся держать оружие**

«Автор»

2026

Примак А.

MINI DOOM современный C++ через игру том 1 учимся держать оружие / А. Примак — «Автор», 2026

Хочешь научиться программировать на C++, но учебники усыпляют на третьей странице? Тогда пиши игру. Это первый том серии «MINI DOOM. Современный C++ через игру», где с нуля, без движков и чужих картинок, создаётся шутер в духе Doom. Здесь — основы современного C++20 в консоли: переменные и типы, условия и циклы, функции и файлы, контейнеры, свои структуры и классы, указатели и умные указатели. Каждая глава заканчивается работающей программой, а финал тома — консольный рогалик с монстрами и аптечками. Весь код собирается со строгими предупреждениями и проверяется санитайзерами, весь вывод в книге — настоящий. Объясняю на «ты», с юмором и без занудства, в каждой главе есть грабли, упражнения и задачи с ответами. Для новичков и тех, кто бросал. 18+.

© Примак А., 2026

© Автор, 2026

Содержание

Предисловие	5
Для кого эта книга	6
Почему Doom	7
Как устроена книга	8
Как читать листинги	9
Цветные врезки	10
Что понадобится	11
Архив примеров	12
Поехали	13
Пять томов	14
Часть I. Учимся держать оружие	15
Глава 1. Hello, Doom	16
Что такое программа и зачем нужен компилятор	16
Ставим Visual Studio 2026	17
Первый проект	17
Настраиваем строгость	18
Hello, Doom	19
Рисуем титульный экран	20
Когда компилятор ругается	22
Итоги главы	25
Упражнения	25
Задачи	26
Глава 2. Переменные и типы	27
Переменная — коробка на складе	27
Какие бывают типы	29
Арифметика и подлость деления	30
Преобразования: когда тип меняется	31
Константы и auto	34
Когда числа ломаются	35
Красивый вывод: std::format	37
Контрольная точка: калькулятор выстрелов	37
Конец ознакомительного фрагмента.	39

Алексей Примак

MINI DOOM современный C++ через игру том 1 учимся держать оружие

Предисловие

Привет. Если ты держишь эту книгу, значит, тебе хочется не просто «выучить C++», а **сделать что-то своё**. Отлично, потому что именно этим мы и займёмся. За двадцать девять глав мы с тобой с нуля напишем игру в духе Doom: с коридорами, дверями, импами, которые швыряются огнём, зомби с винтовками, дробовиком и статус-баром внизу экрана. Без готовых движков, без скачанных картинок и звуков — всё своими руками, строчка за строчкой.



Рисунок 0.1 — Так выглядит MINI DOOM в конце книги: титульный экран, имп вблизи, дробовик, адский зал. Ни одной картинки из интернета — всё нарисовано кодом

Посмотри на рисунок 0.1. Каждый пиксель на нём — результат кода, который ты напишешь. Текстура кирпича, рожа импа, огонь на титульнике и даже звук выстрела — всё это генерируется программой при запуске. Звучит как магия, но к концу книги ты будешь знать, как работает каждый фокус.

Для кого эта книга

Для тех, кто **ни разу не программировал** или пробовал, но бросил на третьей странице скучного учебника. Я не буду требовать от тебя знаний математики сверх школьной, а ту математику, что понадобится — синус, косинус, векторы, — объясню на пальцах прямо тогда, когда она понадобится, и ни минутой раньше.

Если ты уже немного знаешь C++, тоже не уходи. Первую часть можешь пролистать по диагонали, а со второй начнётся то, чего в обычных учебниках нет: своё окно, пиксели, псевдо-3D, искусственный интеллект монстров, синтез звука и многопоточность.

Книга маркирована **18+**. Не потому, что тут кровь и кишки (хотя импы истекают пиксельной кровью), а потому что я разговариваю с тобой как с взрослым: иногда крепким словом, когда программа ведёт себя как последняя сволочь. Дозированно, как соль в супе.

Почему Doom

Потому что Doom — это идеальный учебный проект. Он достаточно простой, чтобы один человек мог написать его за книгу, и достаточно сложный, чтобы по дороге встретилось всё важное: массивы и классы, указатели и память, математика и алгоритмы, потоки и звук. А ещё потому, что стрелять по демонам веселее, чем считать среднюю зарплату сотрудников в отделе, как в половине учебников.

Лирика. Оригинальный Doom вышел в декабре 1993 года. Его написала крошечная студия id Software, а движок сделал Джон Кармак — человек, который выжимал трёхмерную графику из компьютеров, где у процессора не было даже нормальной арифметики с плавающей точкой. Наш MINI DOOM устроен проще настоящего: это рейкастер, ближе к его старшему брату Wolfenstein 3D. Зато мы напишем его на современном C++20, и он будет выдавать сотни кадров в секунду там, где оригинал радовался тридцати пяти.

Как устроена книга

Книга разбита на пять частей. Каждая следующая опирается на предыдущую, поэтому читать лучше по порядку.

Часть	Что изучаем	Что появляется в игре
I. Учимся держать оружие	основы C++: переменные, условия, циклы, функции, контейнеры, классы, память	консольный рогалик: ходим по карте, бьём монстров
II. Окно в ад	своё окно в Windows, пиксели, игровой цикл, клавиатура и мышь, текст	окно, в котором что-то движется и слушается мыши
III. Третье измерение	рейкастинг, текстуры, свет, двери	трёхмерный лабиринт с текстурами и дверями
IV. Здесь кто-то есть	спрайты, оружие, конечные автоматы, поиск пути	монстры, которые охотятся на тебя, и оружие, чтобы им ответить
V. Шум, свет и финальные титры	синтез звука, потоки, HUD, уровни, тесты	законченная игра из двух уровней

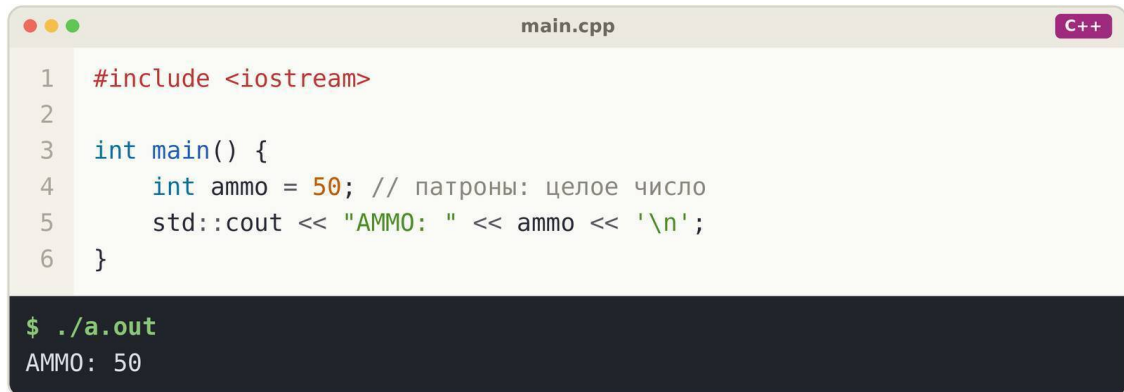
Таблица 0.1 — Пять частей книги

Внутри главы всё идёт по одному сценарию: сначала объясняю идею на пальцах, потом пишем код, потом смотрим на настоящий вывод программы и разбираем его, потом — грабли, на которые наступают все, и в конце упражнения и задачи. Ответы на них собраны в конце книги.

И главное правило: **каждая глава заканчивается работающей программой**. Не «допишем в следующей главе», а прямо сейчас: собрал, запустил, увидел результат. Я называю это контрольной точкой — как в игре, где можно сохраниться.

Как читать листинги

Код в книге выглядит как окно редактора. Сверху — имя файла и язык, слева — номера строк, на которые я ссылаюсь в тексте. Под кодом тёмная панель: это то, что программа напечатала, когда я её запустил. Вот пример — листинг 0.1.



```
1  #include <iostream>
2
3  int main() {
4      int ammo = 50; // патроны: целое число
5      std::cout << "AMMO: " << ammo << '\n';
6  }
```

```
$ ./a.out
AMMO: 50
```

Листинг 0.1 — Как выглядит листинг: код, номера строк и вывод программы

Первая зелёная строка в тёмной панели, «\$./a.out», — это команда запуска. У тебя в Visual Studio никакую команду набирать не нужно: жмёшь Ctrl+F5, и открывается консоль с тем же самым выводом. Строки, начинающиеся с решётки, — мои комментарии, программа их не печатала.

Весь вывод в книге **настоящий**. Каждый пример я компилировал с максимально строгими предупреждениями и проверял санитайзерами — специальными инструментами, которые ловят ошибки работы с памятью. Если у тебя вывод отличается — скорее всего, отличается и код. Сверься с архивом примеров.

Если компилятор отказывается собирать программу, панель подписана «компилятор ругается». Такие листинги я показываю специально: читать ошибки компилятора — такой же навык, как писать код. Сообщения в них от компилятора GCC; Visual Studio скажет то же самое своими словами, и в главе 1 мы разберём, как переводить одно в другое.

Цветные врезки

По тексту разбросаны цветные врезки четырёх видов. Вот они все.

Грабли. Место, где все ломают ноги. Если видишь красную врезку — читай внимательно, тут сэкономишь себе вечер отладки.

Запомни. Короткое правило, которое стоит выучить наизусть. Например: в C++ почти каждая инструкция заканчивается точкой с запятой `;`.

Лирика. Байка, история или шутка. Можно пропустить, но не советую — иногда здесь объясняется, почему всё устроено именно так.

По-взрослому. Как это делают в настоящих проектах, в продакшене. Не обязательно для понимания, но отличает новичка от профи.

Что понадобится

- Компьютер с Windows 10 или 11. Игра использует только то, что есть в самой Windows, никаких сторонних библиотек.
 - Visual Studio 2026, бесплатная редакция Community. Как её поставить, расскажу в главе 1.
 - Желание разобраться. Серьёзно, это важнее всего остального.
- Мышь тоже пригодится: в Doom без мыши играть можно, но это как есть суп вилкой.

Архив примеров

Весь код из книги лежит на GitHub: <https://github.com/RushSex/MiniDoom>. Скачать его можно одним архивом — зелёная кнопка Code, затем Download ZIP — или командой `git clone`; дальше в книге я называю его просто архивом примеров. В папке `examples` там по папке на каждую главу — от `ch01` до `ch29`, — и в каждой папке программа в том виде, в каком она должна быть в конце главы. Если запутался — открой папку своей главы и сравни.

Но не превращай архив в копипасту. Код, набранный своими пальцами, запоминается в десять раз лучше, а опечатки, которые ты при этом наделаешь, научат читать ошибки компилятора. Это не баг обучения, это его фича.

Поехали

Хватит предисловий. Ставим Visual Studio, пишем первую программу и заставляем компьютер сказать «Hello, Doom!». Переворачивай страницу.

Пять томов

Книга «MINI DOOM. Современный C++ через игру» выходит в пяти томах — по одному на каждую часть. Каждый заканчивается работающей программой, и в каждом свои ответы, глоссарий и указатель.

— **Том 1. Учимся держать оружие** — главы 1–8: основы C++ в консоли, от первой программы до памяти и санитайзеров.

— **Том 2. Окно в ад** — главы 9–13: своё окно Windows, пиксели, игровой цикл, клавиатура и мышь, текст.

— **Том 3. Третье измерение** — главы 14–19: рейкастинг, стены, текстуры, пол и свет, двери.

— **Том 4. Здесь кто-то есть** — главы 20–25: спрайты, монстры, оружие, искусственный интеллект, смерть.

— **Том 5. Шум, свет и финальные титры** — главы 26–29: звук, экраны, уровни, CMake и тесты.

Все исходники серии — на GitHub: <https://github.com/RushSex/MiniDoom>. Папка `examples/chNN` для каждой главы, итоговая игра — `examples/ch29/doom`. Там же README с инструкцией по сборке в Visual Studio 2026.

Часть I. Учимся держать оружие

Прежде чем стрелять по демонам, надо научиться держать оружие. В этой части мы освоим основы C++ в самой простой обстановке — в чёрном окне консоли. Никакой графики, никаких окон Windows, только ты, код и компилятор. Зато к концу части у тебя будет консольный рогалик: персонаж бежит по карте, монстры бродят вокруг, а код устроен так, как пишут настоящие программисты.

Глава 1. Hello, Doom

Любое путешествие начинается с первого шага, а любой программист — с программы, которая печатает «Hello, World!». Мы поступим так же, только поздороваемся не с миром, а с Doom. В этой главе ты поставишь Visual Studio, создашь первый проект, настроишь его как профи, заставишь компьютер нарисовать титульный экран игры буквами и, что не менее важно, научишься не бояться, когда компилятор начинает ругаться.

Что такое программа и зачем нужен компилятор

Процессор твоего компьютера — тупой, но очень быстрый исполнитель. Он понимает только машинный код: длинные последовательности чисел вроде «возьми число отсюда, прибавь вот это, положи туда». Писать так руками можно, люди так и делали в пятидесятых, но удовольствия примерно как от заполнения налоговой декларации на латыни.

Поэтому придумали языки программирования. Ты пишешь текст, понятный человеку, а специальная программа — **компилятор** — переводит его в машинный код. Представь шеф-повара, который не знает русского, и переводчика, который берёт твой рецепт и переписывает его на язык повара. Рецепт — это исходный код, переводчик — компилятор, а блюдо — готовая программа, файл с расширением .exe.

На самом деле переводчиков двое. Сначала **компилятор** переводит каждый файл с кодом отдельно и получает объектный файл — полуфабрикат. Потом **компоновщик** (его ещё называют линковщиком) склеивает полуфабрикаты между собой и с готовыми кусками из стандартной библиотеки C++. Получается исполняемый файл. Весь путь показан на рисунке 1.1.



Рисунок 1.1 — Путь от текста программы до файла, который можно запустить

Пока у нас один файл, этот конвейер кажется переусложнённым. Но в настоящей игре файлов десятки, и когда ты поменяешь один из них, перекомпилируется только он, а остальные полуфабрикаты возьмутся готовыми. На больших проектах это экономит часы.

Запомни. Компилятор переводит код в машинные команды и проверяет, что ты не написал ерунды. Компоновщик собирает из кусков готовую программу. Ошибки бывают у обоих, и полезно понимать, кто именно ругается.

Ставим Visual Studio 2026

Visual Studio — это среда разработки: редактор кода, компилятор, компоновщик и отладчик в одной коробке. Для учёбы и личных проектов есть бесплатная редакция **Community**, её нам с головой хватит.

— Скачай установщик Visual Studio 2026 Community с сайта Microsoft и запусти его.

— Установщик спросит, какие «рабочие нагрузки» ставить. Отметь **«Разработка классических приложений на C++»**. В ней компилятор MSVC, библиотеки Windows и всё, что понадобится до конца книги.

— Остальное не трогай и жми «Установить». Скачается несколько гигабайт, так что самое время сделать чай.

Габри. Самая частая беда новичка — поставить Visual Studio без рабочей нагрузки C++. Тогда в списке шаблонов не будет «Консольного приложения» на C++, а при попытке собрать проект появится ошибка про отсутствующий набор инструментов. Лечится так: запусти Visual Studio Installer, нажми «Изменить» и отметь нужную нагрузку.

Лирика. Не путай Visual Studio и Visual Studio Code. Названия похожи, как близнецы, но это разные программы. VS Code — лёгкий текстовый редактор, к которому компилятор надо прикручивать самому. Visual Studio — тяжёлый комбайн, где всё работает из коробки. Нам нужен комбайн.

Первый проект

Запусти Visual Studio. На стартовом экране выбери «Создание проекта». В списке шаблонов найди **«Консольное приложение»** с пометкой C++ — проще всего ввести «консоль» в строку поиска и выбрать язык C++ в фильтре. Нажми «Далее», дай проекту имя, например MiniDoom_ch01, и нажми «Создать».

Visual Studio создаст проект и откроет файл с заготовкой программы. Заготовку можно сразу запустить комбинацией Ctrl+F5: появится консоль с текстом Hello World. Значит, всё установилось правильно. Остальные полезные клавиши собраны в таблице 1.1.

Клавиши	Что делают
Ctrl+F5	собрать и запустить без отладчика; консоль не закроется сама, пока не нажмёшь клавишу
F5	собрать и запустить под отладчиком — пригодится, когда начнём ловить баги
Shift+F5	остановить отладку
Ctrl+Shift+B	только собрать, ничего не запуская
Ctrl+K, Ctrl+C	закомментировать выделенные строки
Ctrl+K, Ctrl+U	раскомментировать их обратно
F12	перейти к определению того, на чём стоит курсор

Таблица 1.1 — Горячие клавиши Visual Studio, которые понадобятся сразу

Настраиваем строгость

По умолчанию компилятор ведёт себя как добрый учитель: видит подозрительное место, вздыхает, пишет предупреждение мелким шрифтом и всё равно собирает программу. Предупреждения тонут в выводе, ты их не читаешь, а через неделю программа падает ровно там, где компилятор вежливо намекал. Мы сделаем из него злого прапорщика.

Щёлкни правой кнопкой по проекту в «Обозревателе решений» и выбери «Свойства». Вверху окна поставь «Конфигурация: Все конфигурации» и «Платформа: Все платформы», чтобы настройки применились и к отладочной, и к финальной сборке. Затем выставь значения из таблицы 1.2.

Где найти	Что поставить	Зачем
C/C++ → Язык → Стандарт языка C++	ISO C++20 (/std:c++20)	пишем на современном C++
C/C++ → Общие → Уровень предупреждений	Уровень 4 (/W4)	компилятор замечает максимум подозрительного
C/C++ → Общие → Обращаться к предупреждениям как ошибки	Да (/WX)	любое предупреждение останавливает сборку
C/C++ → Язык → Режим совместимости	Да (/permissive-)	строгое следование стандарту
C/C++ → Командная строка → Дополнительные параметры	/utf-8	файлы с кодом читаются в UTF-8, русские комментарии не превращаются в кашу

Таблица 1.2 — Настройки проекта, которые мы ставим в каждой главе

По-взрослому. В серьёзных командах сборка с `/W4 /WX` — норма, а не прихоть. Предупреждение сегодня — это баг завтра. Когда код собирается начисто, любое новое предупреждение сразу бросается в глаза, а не прячется среди сотни старых, на которые все давно забыли.

Hello, Doom

Удали из файла всё, что там написала Visual Studio, и набери код из листинга 1.1. Именно набери руками, не копируй: пальцы тоже должны учиться.

```

1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, Doom!\n";
5      return 0;
6  }

```

\$./a.out
Hello, Doom!

Листинг 1.1 — Первая программа

Нажми `Ctrl+F5`. Откроется консоль, и в ней будет ровно то, что в тёмной панели под кодом: строка `Hello, Doom!`. Поздравляю, ты программист. Ну, почти. Теперь разберём, что тут происходит, строчка за строчкой.

Строка 1. `#include <iostream>` подключает кусок стандартной библиотеки, который умеет печатать текст и читать с клавиатуры. Название расшифровывается как `input/output stream` — поток ввода-вывода. Решётка в начале означает, что это команда не для компилятора, а для его помощника — препроцессора. Он буквально вставляет на это место содержимое файла `iostream`, как если бы ты скопировал его руками.

Строка 3. `int main()` — это функция `main`, точка входа в программу. С неё всё начинается, когда ты запускаешь `.exe`. Слово `int` перед ней означает, что функция вернёт целое число, а пустые круглые скобки — что ей ничего не передают. Фигурные скобки на строках 3 и 6 отмечают начало и конец тела функции.

Строка 4. Главная. `std::cout` — это консоль, точнее поток вывода в неё. Префикс `std::` говорит, что `cout` живёт в пространстве имён `std` — там лежит вся стандартная библиотека. Оператор `<<` можно читать как «отправь туда»: строку `"Hello, Doom!\n"` отправили в консоль. Два символа `\n` в конце — это перевод строки: курсор переходит на новую строку, как от `Enter`. Точка с запятой в конце завершает инструкцию.

Строка 5. `return 0;` завершает функцию `main` и сообщает операционной системе: всё прошло нормально. Ноль по старой традиции означает успех, любое другое число — какую-то ошибку. У `main` есть привилегия: если `return` забыть, C++ сам вернёт 0. Но мы пока пишем явно, чтобы было видно.

Как оператор `<<` гонит данные в консоль, показано на рисунке 1.2. Его удобно представлять конвейерной лентой: что положил слева, то и уехало направо, в том же порядке.

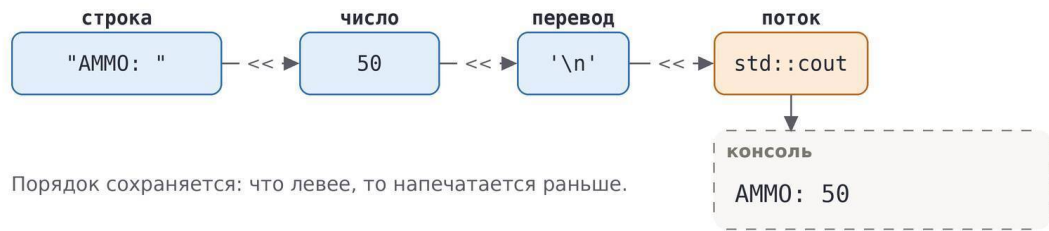


Рисунок 1.2 — Оператор `<<` как конвейер: значения по очереди уезжают в `std::cout` и появляются в консоли

Запомни. Почти каждая инструкция в C++ заканчивается точкой с запятой `;`. Компилятору плевать на переносы строк и пробелы, для него конец мысли — именно `;`.

Рисуем титульный экран

Строка `Hello, Doom!` — это мило, но у настоящей игры должен быть титульный экран. Нарисуем его буквами, как делали в восьмидесятых, — это называется ASCII-арт. Код в листинге 1.2.

```

main.cpp
1 // title.cpp – титульный экран нашей будущей игры
2 #include <iostream>
3
4 int main() {
5     // R"(...)" – «сырая» строка: обратные слэши и переносы
6     // строк попадают в неё как есть, ничего не надо экранировать.
7     std::cout << R"(
8
9     |  \  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |
10    |  \  |  |  |  \  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
11    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
12    |  |  |  |  |  \  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
13 );";
14     std::cout << "\n          C++20 RAYCASTER * PRESS ENTER\n";
15     return 0;
16 }

```

```

$ ./a.out

```

```

|  \  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  |  _  | | | | | | | | | | |
|  \  |  |  |  \  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  \  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

```

```

          C++20 RAYCASTER * PRESS ENTER

```

Листинг 1.2 — Титульный экран *MINI DOOM* из символов

Сравни код и вывод: картинка напечаталась ровно в том виде, в каком лежит в коде. Всё благодаря «сырой» строке `R"(...)"` на строках 7–13. В обычной строке обратный слэш — особый символ: `\n` означает перевод строки, а чтобы напечатать сам слэш, его надо удваивать. В нашем рисунке слэшей полно, и с обычной строкой пришлось бы аккуратно удваивать каждый. Сырая строка отключает эту магию: всё между `R" ( и )"` попадает в программу буква в букву, вместе с переносами строк.

Строки 1, 5 и 6 начинаются с двух косых черт. Это **комментарии**: всё от `//` до конца строки компилятор пропускает. Комментарии пишут для людей — для коллег и для себя через полгода, когда забудешь, что тут происходит. А забудешь обязательно.

Обрати внимание на первую строку вывода: она пустая. Это не баг. Сырая строка начинается сразу после открывающей скобки, а там стоит перевод строки — он тоже попал в вывод.

Грабли. Захочешь написать на титульнике что-нибудь по-русски — скорее всего, увидишь в консоли кракозябры. Консоль Windows по историческим причинам может работать не в той кодировке, в которой записан твой текст. Лечится это настройкой и консоли, и компилятора, и мы разберёмся с этим позже. А в части II у нас будет своё окно со своим шрифтом, и проблема исчезнет сама. Поэтому пока все надписи в игре — по-английски, как в оригинальном Doom.

Когда компилятор ругается

Рано или поздно — скорее рано — ты нажмёшь Ctrl+F5, а вместо программы получишь список ошибок. Это нормально. Опытные программисты видят ошибки компилятора по сто раз на дню и не расстраиваются, потому что ошибка компиляции — лучшая ошибка из возможных. Её нашли до запуска, а не пользователь посреди уровня.

Давай сломаем нашу программу специально и посмотрим, что скажет компилятор. Начнём с классики — забытой точки с запятой в листинге 1.3.

```

1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, Doom!\n"
5      return 0;
6  }

```

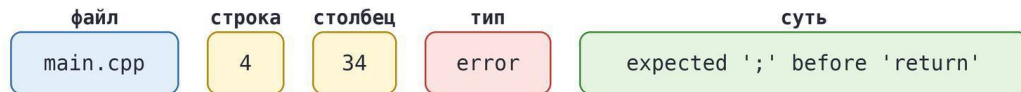
```

$ компилятор ругается:
main.cpp: In function 'int main()':
main.cpp:4:34: error: expected ';' before 'return'
   4 |         std::cout << "Hello, Doom!\n"
     |                                     ^
     |                                     ;
   5 |         return 0;
     |         ~~~~~

```

Листинг 1.3 — Забыли точку с запятой после строки 4

Сообщение выглядит пугающе, но устроено всегда одинаково. Разберём его по частям на рисунке 1.3.



Читай справа налево: суть → где → в каком файле.

Ожидалась ';' перед 'return' — значит, её забыли в конце предыдущей строки.

Рисунок 1.3 — Анатомия сообщения об ошибке: файл, строка, столбец, тип и суть

Компилятор говорит: в файле `main.cpp`, в строке 4, в столбце 34 он ожидал точку с запятой перед словом `return`. Ниже он даже рисует, куда её поставить. Добавь ; в конец строки 4 — и всё соберётся.

Грабли. Компилятор часто указывает не на строку с ошибкой, а на следующую. Он читает код подряд и спотыкается только тогда, когда встречает что-то, что уже точно не лезет. Точку с запятой забыли в строке 4, но понял он это, только увидев `return` в строке 5. Если строка из сообщения выглядит правильной — смотри на строку выше.

Вторая классика — опечатка. В листинге 1.4 вместо `cout` написано `cot`.

```

main.cpp
C++
1  #include <iostream>
2
3  int main() {
4      std::cot << "Hello, Doom!\n";
5      return 0;
6  }

$ компилятор ругается:
main.cpp: In function 'int main()':
main.cpp:4:10: error: 'cot' is not a member of 'std'; did you mean 'oct'?
    4 |         std::cot << "Hello, Doom!\n";
      |         ^~~
      |         oct
    
```

Листинг 1.4 — Опечатка в имени: `cot` вместо `cout`

Компилятор честно говорит, что в `std` нет никакого `cot`, и даже пытается угадать, что ты имел в виду. Угадывает, правда, `oct` — это штука для печати чисел в восьмеричной системе, к нашей ситуации отношения не имеет. Подсказки компилятора полезны, но слепо им верить не стоит: он видит буквы, а не твои намерения.

Третий пример хитрее. В листинге 1.5 синтаксически всё правильно, но есть переменная `health`, которой мы так и не воспользовались. Переменные разберём в следующей главе, а пока просто посмотри, что происходит.

```

main.cpp
1  #include <iostream>
2
3  int main() {
4      int health = 100;
5      std::cout << "Hello, Doom!\n";
6      return 0;
7  }

$ компилятор ругается:
main.cpp: In function 'int main()':
main.cpp:4:9: error: unused variable 'health' [-Werror=unused-variable]
    4 |     int health = 100;
      |         ^~~~~~
cc1plus: all warnings being treated as errors

```

Листинг 1.5 — Переменная объявлена, но не используется

Без строгих настроек это было бы безобидное предупреждение, и программа собралась бы. Но мы включили режим «предупреждения как ошибки», поэтому сборка остановилась. Последняя строка так и говорит: все предупреждения считаются ошибками. Зачем нам такие сложности? Неиспользуемая переменная часто означает, что ты что-то забыл. Например, посчитал урон и забыл его нанести.

Сообщения в листингах — от компилятора GCC. Visual Studio скажет то же самое своими словами и с номером ошибки, который удобно гуглить. Соответствие собрано в таблице 1.3.

Что пишет GCC	Код в Visual Studio	Что случилось
expected ';' before 'return'	C2143	забыта точка с запятой — ищи в конце предыдущей строки
'cot' is not a member of 'std'	C2039	опечатка в имени из стандартной библиотеки
'x' was not declared in this scope	C2065	имя не объявлено: опечатка или забыт #include
unused variable 'health'	C4189	переменная создана, но не используется
all warnings being treated as errors	C2220	сработал режим /WX: предупреждение остановило сборку

Таблица 1.3 — Как переводить ошибки GCC на язык Visual Studio

Главные правила чтения ошибок такие:

- Читай **первую** ошибку. Одна настоящая ошибка часто тянет за собой десяток навешенных, и после её исправления остальные исчезают сами.
- Смотри на номер строки и на строку выше.

— Читай суть целиком, а не только слово `error`. Компилятор обычно говорит прямо, чего ему не хватает.

— Если ничего не понятно — скопируй код ошибки и поищи его в интернете. С этой ошибкой до тебя уже встретились тысячи людей.

Итоги главы

— Компилятор переводит код в машинные команды, компоновщик склеивает из них программу.

— Visual Studio 2026 с нагрузкой «Разработка классических приложений на C++» — наш верстак до конца книги.

— Каждый проект настраиваем строго: `C++20, /W4, /WX, /permissive-, /utf-8`.

— Программа начинается с функции `main`, текст печатается через `std::cout` и оператор `<<`.

— Сырая строка `R"(...)"` сохраняет текст как есть — идеально для ASCII-арта.

— Ошибки компилятора не страшны: читаем первую, смотрим на строку и строку выше.

Контрольная точка: у тебя есть программа, которая печатает титульный экран MINI DOOM. В следующей главе на нём появятся числа — здоровье, патроны и урон, — и мы посчитаем, сколько выстрелов нужно, чтобы уложить импа.

Упражнения

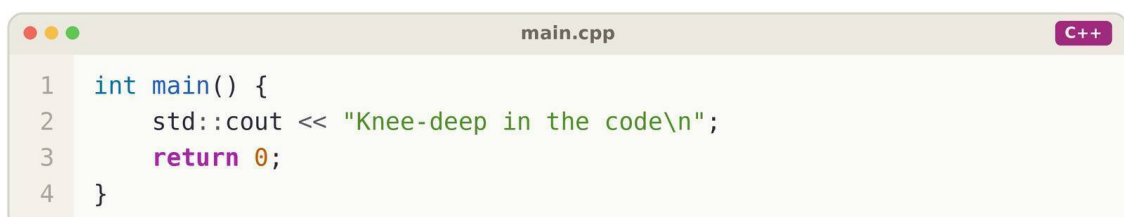
Упражнение 1.1. *Что выведет?* Не запуская программу из листинга 1.6, напиши на бумаге, что она напечатает. Потом проверь себя.



```
main.cpp C++
1  #include <iostream>
2
3  int main() {
4      std::cout << "RIP";
5      std::cout << " AND ";
6      std::cout << "TEAR\n";
7      std::cout << "\n";
8      std::cout << "LEVEL " << 1 << '\n';
9      std::cout << 2 + 2 << " = " << "2 + 2\n";
10 }
```

Листинг 1.6 — Упражнение 1.1

Упражнение 1.2. *Найди баг.* Программа из листинга 1.7 не собирается. Почему? Попробуй догадаться до того, как соберёшь её.



```
main.cpp C++
1  int main() {
2      std::cout << "Knee-deep in the code\n";
3      return 0;
4  }
```

Листинг 1.7 — Упражнение 1.2

Упражнение 1.3. *Напиши сам.* Выведи надпись PRESS ENTER в рамке из символов +, - и |. Рамка должна быть ровной: углы на своих местах, стороны одной длины.

Упражнение 1.4. *Подумай.* Программа с неиспользуемой переменной из листинга 1.5 прекрасно работала бы и без режима /WX. Зачем же мы сами себе мешаем? Приведи пример, когда лишнее предупреждение спасает от настоящего бага.

Задачи

Задача 1.1. Экран загрузки. Напиши программу, которая печатает заголовок с версией игры и несколько строк вида «Loading textures ОК», где слово ОК во всех строках стоит строго в одном столбце. Закончи боевым кличем.

Задача 1.2. Свой баннер. Нарисуй в сырой строке титульник со своим ником. Потом попробуй вставить в рисунок сочетание символов)" — например, нарисуй смайлик. Что случится и как это обойти? Подсказка: у сырой строки можно задать свой разделитель.

Задача 1.3. Коллекционер ошибок. Сломай программу из листинга 1.1 пятью разными способами: убери скобку, закавычь не то, переименуй main, удали #include и так далее. Для каждой поломки выпиши первую строку ошибки и объясни её своими словами. Это лучшая прививка от страха перед компилятором.

Глава 2. Переменные и типы

Открою тебе страшную тайну: любая игра — это просто куча чисел. Здоровье игрока — число. Патроны — число. Координаты импа, угол поворота камеры, урон дробовика, время на уровне — числа, числа, числа. Картинка на экране и звук в колонках — тоже числа, только их очень много. Поэтому первое, чему надо научиться, — хранить числа и считать с ними, не отстрелив себе ногу.

В этой главе мы разберёмся, что такое переменная, какие бывают типы, почему $7 / 2$ в C++ равно трём, а $0,1 + 0,2$ не равно $0,3$. А в конце напишем калькулятор, который посчитает, сколько выстрелов из пистолета нужно, чтобы уложить импа.

Переменная — коробка на складе

Представь огромный склад, где вдоль стен стоят пронумерованные коробки. Номер коробки — это **адрес** в памяти. На коробку можно наклеить этикетку с именем, например `health`, и положить внутрь число 100. Это и есть **переменная**: именованная коробка в памяти, в которой лежит значение.

У каждой коробки есть ещё и **тип** — он говорит, что в ней можно хранить и какого она размера. В коробку для целых чисел не положишь дробное, а в коробку для одной буквы не влезет целое предложение. Посмотрим на листинг 2.1.



```

1  #include <iostream>
2
3  int main() {
4      int health = 100;    // здоровье игрока
5      int ammo = 50;      // патроны
6      double speed = 3.4; // клеток в секунду
7
8      std::cout << "HEALTH: " << health << '\n';
9      std::cout << "AMMO:   " << ammo << '\n';
10     std::cout << "SPEED:  " << speed << '\n';
11
12     ammo = ammo - 1;    // выстрелили
13     health -= 15;       // в нас попали
14     std::cout << "after: " << health << " HP, " << ammo << " ammo\n";
15 }

```

```

$ ./a.out
HEALTH: 100
AMMO:   50
SPEED:  3.4
after: 85 HP, 49 ammo

```

Листинг 2.1 — Переменные игрока: создаём, печатаем, меняем

Строки 4–6 создают три переменные. Запись `int health = 100;` читается так: «заведи коробку типа `int` с этикеткой `health` и положи в неё 100». Тип `int` хранит целые числа, `double`

— дробные. Здоровье и патроны у нас целые, а скорость — 3,4 клетки в секунду — дробная. Заметь: в коде дробную часть отделяют точкой, а не запятой.

Строки 8–10 печатают значения. Когда в `std::cout` отправляешь имя переменной, печатается не имя, а то, что лежит в коробке.

Строка 12 самая интересная. `ammo = ammo - 1`; с точки зрения математики — бред: число не может быть равно самому себе минус один. Но в C++ знак `=` означает не «равно», а «положи»: возьми, что лежит в `ammo`, вычти единицу и положи результат обратно в `ammo`. Было 50 — стало 49.

Строка 13 делает то же самое короче: `health -= 15`; значит «уменьши `health` на 15». Такие сокращения есть для всех операций: `+=`, `-=`, `*=`, `/=`. Вывод подтверждает: после стычки у нас 85 здоровья и 49 патронов.

Запомни. Знак `=` в C++ — это не «равно», а «положить в коробку». Слева всегда коробка, справа — то, что в неё кладём. Сравнение на равенство пишется двумя знаками: `==`, до него дойдём в главе 3.

Хочешь посмотреть на склад своими глазами? Оператор `&` перед именем переменной выдаёт её адрес, а `sizeof` — размер в байтах. Листинг 2.2 печатает и то, и другое; буква `B` в выводе — байты.

```

main.cpp C++
1  #include <iostream>
2
3  int main() {
4      int health = 100;
5      double speed = 3.4;
6      char key = 'W';
7      bool alive = true;
8
9      std::cout << "health: " << sizeof(health) << " B at "
10         << &health << '\n';
11      std::cout << "speed:  " << sizeof(speed) << " B at "
12         << &speed << '\n';
13      std::cout << "key:     " << sizeof(key) << " B at "
14         << static_cast<void*>(&key) << '\n';
15      std::cout << "alive:  " << sizeof(alive) << " B at "
16         << &alive << '\n';
17  }

$ ./a.out
health: 4 B at 0x7fa568800050
speed:  8 B at 0x7fa568800060
key:    1 B at 0x7fa568800030
alive:  1 B at 0x7fa568800040

```

Листинг 2.2 — Где живут переменные и сколько места занимают

Адреса печатаются в шестнадцатеричной системе — поэтому там буквы. У тебя они будут другими, и даже у меня при каждом запуске меняются: операционная система специально раскладывает программу по памяти случайно, чтобы злоумышленникам было сложнее. А вот размеры будут те же: `int` занимает 4 байта, `double` — 8, `char` и `bool` — по одному. Как это выглядит на складе, показано на рисунке 2.1.



Рисунок 2.1 — Переменные из листинга 2.2 на складе памяти: имя, значение, адрес и размер

Какие бывают типы

Типов в C++ много, но для игры на первых порах хватит шести. Листинг 2.3 печатает их размеры и пределы. `std::numeric_limits` — это справочник стандартной библиотеки: спрашиваешь у него про тип, он отвечает, какое самое большое число туда влезает и сколько знаков точности он держит.

```

main.cpp C++
1  #include <iostream>
2  #include <limits>
3
4  int main() {
5      std::cout << "bool      " << sizeof(bool) << " byte\n";
6      std::cout << "char      " << sizeof(char) << " byte\n";
7      std::cout << "int       " << sizeof(int) << " bytes, max "
8          << std::numeric_limits<int>::max() << '\n';
9      std::cout << "long long " << sizeof(long long) << " bytes, max "
10         << std::numeric_limits<long long>::max() << '\n';
11      std::cout << "float      " << sizeof(float) << " bytes, ~"
12         << std::numeric_limits<float>::digits10 << " digits\n";
13      std::cout << "double     " << sizeof(double) << " bytes, ~"
14         << std::numeric_limits<double>::digits10 << " digits\n";
15  }

$ ./a.out
bool      1 byte
char      1 byte
int       4 bytes, max 2147483647
long long 8 bytes, max 9223372036854775807
float     4 bytes, ~6 digits
double    8 bytes, ~15 digits

```

Листинг 2.3 — Размеры и пределы основных типов

Обрати внимание на `int`: максимум чуть больше двух миллиардов. Для здоровья и патронов — с огромным запасом. А вот `float` держит всего около шести значащих цифр, `double` — около пятнадцати. Поэтому для дробных чисел по умолчанию бери `double`, а `float` — только когда памяти жалко или так требует графика. Что где пригодится в игре, собрано в таблице 2.1.

Тип	Что хранит	Пример в игре
<code>int</code>	целое число, примерно ± 2 миллиарда	здоровье, патроны, убитые монстры
<code>long long</code>	очень большое целое	очки за всю карьеру, время в миллисекундах
<code>double</code>	дробное число, ~15 цифр	скорость, расчёт среднего урона
<code>float</code>	дробное число, ~6 цифр, вдвое меньше	координаты на карте в настоящей игре
<code>bool</code>	true или false	жив ли монстр, открыта ли дверь
<code>char</code>	один символ	клетка карты: стена # или пол .

Таблица 2.1 — Основные типы и где они пригодятся в MINI DOOM

Грабли. Есть ещё тип `long`, и он подлый. В Visual Studio под Windows он занимает 4 байта, как `int`, а в Linux — 8. Код, который у тебя работал, у друга на другой системе внезапно переполнится. Поэтому `long` в этой книге мы не используем вообще: нужно больше, чем у `int`, — бери `long long`.

Арифметика и подлость деления

Сложение, вычитание и умножение в C++ работают так, как ты ожидаешь. А вот деление приготовило сюрприз. Смотри листинг 2.4.

```

main.cpp C++
1  #include <iostream>
2
3  int main() {
4      int hp = 60;        // здоровье импа
5      int damage = 15;    // урон за выстрел
6
7      std::cout << "60 / 15 = " << hp / damage << '\n';
8      std::cout << "60 / 7 = " << hp / 7 << '\n';    // целое деление!
9      std::cout << "60 % 7 = " << hp % 7 << '\n';    // остаток
10     std::cout << "60.0 / 7 = " << 60.0 / 7 << '\n';
11     std::cout << "7 / 2 * 2 = " << 7 / 2 * 2 << '\n';
12 }

$ ./a.out
60 / 15 = 4
60 / 7 = 8
60 % 7 = 4
60.0 / 7 = 8.57143
7 / 2 * 2 = 6

```

Листинг 2.4 — Целое деление и остаток

$60 / 15 = 4$ — пока всё честно. Но $60 / 7$ дало 8, хотя на калькуляторе будет 8,57. Дело в том, что если **оба** числа целые, C++ делит нацело и просто отбрасывает дробную часть. Не округляет, а именно отрезает: даже 8,99 превратилось бы в 8.

Остаток от деления даёт оператор `%`: $60 \% 7 = 4$, потому что $7 \times 8 = 56$, и до 60 не хватает четырёх. Звучит как школьная скукоптица, но в играх остаток нужен постоянно: перевести секунды в минуты, сделать анимацию, которая крутится по кругу, проверить чётность.

Стоит хотя бы одному числу стать дробным — 60.0 вместо 60, — и деление становится обычным: 8.57143. А последняя строка — классическая ловушка: $7 / 2 * 2$ даёт не 7, а 6. Сначала $7 / 2$ нацело = 3, потом $3 \times 2 = 6$. Половинка потерялась на первом шаге и больше не вернулась.

Габбли. `int / int` — это всегда целое деление. Дробь отрезается молча, без единого предупреждения, ведь с точки зрения компилятора ты сам этого хотел. Если нужен дробный результат, сделай дробным хотя бы одно из чисел: `60.0 / 7` или `static_cast<double>(hp) / 7`.

Преобразования: когда тип меняется

Иногда значение одного типа надо положить в коробку другого. Например, средний урон 9,75 превратить в целое число патронов. Компилятор такие переходы не любит, потому что дробная часть при этом теряется. Есть два способа создать переменную: со знаком `=` и с фигурными скобками. Начнём со скобок — листинг 2.5.

```

main.cpp
1  int main() {
2      double average = 9.75;
3      int ammo{average}; // скобки не дадут потерять дробь
4      return ammo;
5  }

$ компилятор ругается:
main.cpp: In function 'int main()':
main.cpp:3:14: error: narrowing conversion of 'average' from 'double' to
      'int' [-Werror=narrowing]
      3 |         int ammo{average}; // скобки не дадут потерять дробь
        |                   ^~~~~~
main.cpp:3:14: error: conversion from 'double' to 'int' may change value
      [-Werror=float-conversion]
cc1plus: all warnings being treated as errors

```

Листинг 2.5 — Фигурные скобки запрещают сужение

Запись `int ammo{average};` — это **инициализация в фигурных скобках**. Она строже: если значение не влезает в тип без потерь, это ошибка сразу, по правилам языка, без всяких настроек. Компилятор называет это `narrowing conversion` — сужающее преобразование, когда из большей коробки пытаются пересыпать в меньшую.

А что со знаком `=`? В листинге 2.6 то же самое, но по-старому.

```

main.cpp
1  int main() {
2      double average = 9.75;
3      int ammo = average; // строгий режим не пустит и так
4      return ammo;
5  }

$ компилятор ругается:
main.cpp: In function 'int main()':
main.cpp:3:16: error: conversion from 'double' to 'int' may change value
      [-Werror=float-conversion]
      3 |         int ammo = average; // строгий режим не пустит и так
        |                   ^~~~~~
cc1plus: all warnings being treated as errors

```

Листинг 2.6 — Со знаком `=` потерю дробки ловит строгий режим

По правилам языка это разрешено: дробь молча отрежется, и в `ammo` будет 9. Поймал это только наш строгий режим с предупреждениями, превращёнными в ошибки. Visual Studio выдаст тут предупреждение C4244, а с `/WX` — ошибку. Без строгих настроек такой код собрался бы, и ты потом неделю искал бы, куда пропадают патроны.

Когда отрезать дробь — именно то, что нужно, скажи об этом компилятору явно через `static_cast`. В листинге 2.7 заодно видна самая популярная ошибка с процентами.

```

main.cpp C++
1  #include <iostream>
2
3  int main() {
4      int kills = 3;
5      int total = 14;
6
7      std::cout << "wrong: " << kills / total * 100 << "%\n";
8      std::cout << "int:   " << kills * 100 / total << "%\n";
9      const double exact = static_cast<double>(kills) / total * 100;
10     std::cout << "exact: " << exact << "%\n";
11
12     double time = 83.9;           // секунд на уровне
13     int seconds = static_cast<int>(time); // дробь отрезается
14     std::cout << "seconds: " << seconds << '\n';
15 }

$ ./a.out
wrong: 0%
int:   21%
exact: 21.4286%
seconds: 83

```

Листинг 2.7 — Проценты убитых монстров и явное приведение типов

Первая строка вывода — 0%, хотя мы убили троих из четырнадцати. Почему так получилось, видно на рисунке 2.2: выражение считается слева направо, и $3 / 14$ нацело — это ноль. Ноль умножить на сто — ноль. Если сначала умножить на 100, а потом делить, получится 21% — уже похоже на правду. А `static_cast<double>(kills)` превращает `kills` в дробное число на время вычисления, и мы получаем точные 21,4286%.

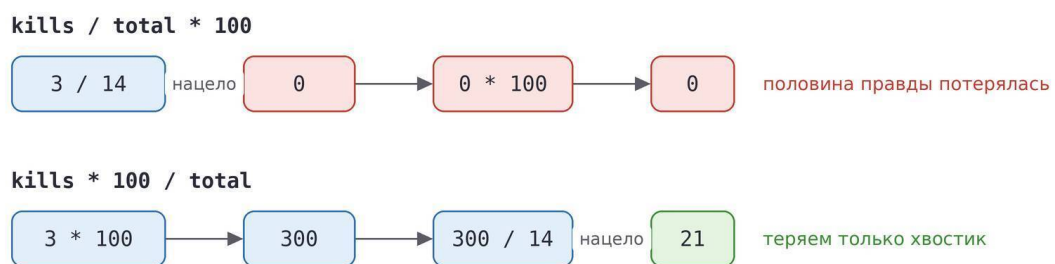


Рисунок 2.2 — Порядок действий решает: деление нацело в начале убивает результат, в конце — только хвостик

Последние строки листинга показывают обратное превращение: `static_cast<int>(time)` отрезает дробную часть, и из 83,9 секунды остаётся 83. Не 84 — именно отрезает, а не округляет. Запомни это, пригодится в главе 27, когда будем показывать время прохождения уровня.

Константы и auto

Здоровье импа в Doom — 60 единиц, и оно не меняется посреди игры. Если в коде в двадцати местах написано просто 60, то, во-первых, непонятно, что это за число, а во-вторых, когда захочешь сделать импов покрепче, придётся найти и исправить все двадцать мест — и пропустить хотя бы одно. Такие числа программисты зовут **магическими** и не любят. Лекарство — в листинге 2.8.



```

1  #include <iostream>
2
3  constexpr int kImpHealth = 60;    // не меняется никогда
4  constexpr int kPistolMin = 5;
5  constexpr int kPistolMax = 15;
6
7  int main() {
8      auto ammo = 50;               // auto: тип выведется из 50 — int
9      const auto half = ammo / 2;
10     std::cout << "imp: " << kImpHealth << ", half: " << half << '\n';
11     // kImpHealth = 100;           // раскомментируй — не соберётся
12     std::cout << "dmg: " << kPistolMin << ".." << kPistolMax << '\n';
13 }

```

```

$ ./a.out
imp: 60, half: 25
dmg: 5..15

```

Листинг 2.8 — Константы `constexpr`, `const` и автоматический `min auto`

Строки 3–5. `constexpr int kImpHealth = 60;` создаёт константу, которая известна уже во время компиляции. Её нельзя изменить, а компилятор подставит значение прямо в машинный код. Букву `k` в начале имени я ставлю по традиции — сразу видно, что это константа.

Строка 8. `auto ammo = 50;` — компилятор сам выводит тип из значения. Пятьдесят — целое, значит, `ammo` будет `int`. `auto` экономит время, когда тип длинный и очевидный, но злоупотреблять не стоит: в коде должно быть понятно, что где лежит.

Строка 9. `const` тоже запрещает изменение, но значение может быть известно только при запуске программы. Половину патронов мы посчитали из переменной `ammo`, поэтому тут `const`, а не `constexpr`.

Строка 11. Попытка изменить константу. Сейчас строка закомментирована; раскомментируй — и компилятор откажется собирать программу, потому что менять константы нельзя. Это и есть их смысл: компилятор сторожит то, что ты сам объявил неизменным.

По-взрослому. Хорошее правило: всё, что не обязано меняться, объявляй `const` или `constexpr`. Тогда при чтении кода сразу видно, какие переменные «живые», а какие нет, а случайное изменение константы компилятор поймает за руку.

Когда числа ломаются

У каждой коробки есть дно и крышка. Что будет, если в `int`, где и так лежит самое большое возможное число, добавить единицу? Проверим в листинге 2.9. Этот пример я собрал с санитайзером — специальным сторожем, который следит за программой во время работы.

```

1  #include <iostream>
2  #include <limits>
3
4  int main() {
5      int score = std::numeric_limits<int>::max(); // самое большое int
6      std::cout << "score: " << score << '\n';
7      score = score + 1;                          // а теперь ещё очко
8      std::cout << "score: " << score << '\n';
9  }

$ ./a.out # собрано с -fsanitize=address,undefined
score: 2147483647
main.cpp:7:11: runtime error: signed integer overflow: 2147483647 + 1
cannot be represented in type 'int'

```

Листинг 2.9 — Переполнение `int`: санитайзер ловит его на месте

Первая строка напечаталась нормально. А на строке 7 санитайзер схватил программу за руку: знаковое целое переполнилось, $2147483647 + 1$ не помещается в `int`. Программа остановилась прямо на месте преступления, и вторая строка так и не напечаталась.

А что было бы без сторожа? Соберём тот же код без санитайзера и запустим — листинг 2.10.

```

# собираем без санитайзера
$ g++ -std=c++20 main.cpp -o doom
$ ./doom
score: 2147483647
score: -2147483648

```

Листинг 2.10 — Без санитайзера переполнение проходит молча

Число перевалило через край и превратилось в самое маленькое отрицательное — как старый автомобильный одометр, который после 99999 показывает 00000. Только хуже: по стандарту C++ переполнение знакового целого — это **неопределённое поведение**. Компилятор вправе считать, что его никогда не бывает, и на этом основании выкидывать из программы проверки. Сегодня у тебя «заворачивается», завтра с другими настройками оптимизации происходит что-то совсем странное.

Грабли. В Visual Studio есть AddressSanitizer — сторож для ошибок памяти, мы включим его в главе 8. А вот сторожа для переполнения целых, как в листинге выше, у компилятора MSVC нет. Значит, думать о пределах типов придётся головой: если число может вырасти больше двух миллиардов, сразу бери `long long`.

С дробными числами своя беда — они не точные. Листинг 2.11 проверяет, равно ли $0,1 + 0,2$ трём десяткам.

```

1  #include <format>
2  #include <iostream>
3
4  int main() {
5      double a = 0.1 + 0.2;
6      std::cout << std::format("{}\n", a == 0.3);
7      std::cout << std::format(":{:.17f}\n", a);
8      std::cout << std::format(":{:.17f}\n", 0.3);
9  }

```

```

$ ./a.out
false
0.30000000000000004
0.29999999999999999

```

Листинг 2.11 — Дробные числа хранятся приблизительно

Ответ — `false`, не равно. Компьютер хранит дробные числа в двоичной системе, а там $0,1$ — бесконечная дробь, как $1/3$ в десятичной. Она обрезается, и при печати с семнадцатью знаками видно, что $0,1 + 0,2$ чуть больше $0,3$, а само $0,3$ чуть меньше. Разница крохотная, но для `==` её достаточно. Здесь я печатал через `std::format` — о нём чуть ниже.

Запомни. Никогда не сравнивай дробные числа через `==`. Сравнивай с допуском: «разница меньше одной миллионной». В нашей игре мы так и будем делать, например, когда проверим, дошёл ли монстр до цели.

Лирика. В 1993 году у многих компьютеров не было блока для быстрой работы с дробными числами, поэтому Кармак в Doom обошёлся вообще без них. Координаты хранились в целых числах с фиксированной точкой: старшие 16 бит — целая часть, младшие 16 — дробная. Мы такими извращениями заниматься не будем: современный процессор считает `double` так же быстро, как `int`.

Красивый вывод: `std::format`

Помнишь задачу из главы 1, где точки для выравнивания пришлось считать руками? В C++20 для этого есть `std::format` из заголовка `<format>`. Он собирает строку по шаблону: в фигурные скобки подставляются значения, а после двоеточия можно указать, как именно их печатать. Самые нужные варианты — в таблице 2.2.

Шаблон	Что делает	Пример
<code>{}</code>	просто подставить значение	<code>format("{} HP", 85) → 85 HP</code>
<code>{:>6}</code>	выровнять вправо в поле шириной 6	<code>" 60"</code>
<code>{:<22}</code>	выровнять влево в поле шириной 22	надписи в столбик
<code>{:.1f}</code>	дробное с одним знаком после точки	<code>6.0</code>
<code>{:02}</code>	целое минимум в 2 знака, слева нули	<code>05</code> для 1:05

Таблица 2.2 — Самые полезные шаблоны `std::format`

Строка формата проверяется ещё при компиляции: если в шаблоне три пары скобок, а значений два, программа не соберётся. Это одна из причин, почему в новом коде `std::format` лучше, чем старые способы.

Контрольная точка: калькулятор выстрелов

Соберём всё вместе. Пистолет в Doom наносит от 5 до 15 урона за выстрел, у импа 60 здоровья. Сколько выстрелов понадобится при большом везении, при полном невезении и в среднем? А сколько урона в среднем наносит дробовик, если в его патроне семь дробин? Ответы — в листинге 2.12.

```

main.cpp C++
1 // calc.cpp — сколько выстрелов нужно, чтобы уложить импа
2 #include <format>
3 #include <iostream>
4
5 constexpr int kImpHealth = 60;
6 constexpr int kPistolMin = 5; // урон пистолета: от 5
7 constexpr int kPistolMax = 15; // до 15
8 constexpr int kPellets = 7; // дробинки в патроне
9
10 int main() {
11     // Деление с округлением вверх в целых: (a + b - 1) / b.
12     const int best = (kImpHealth + kPistolMax - 1) / kPistolMax;
13     const int worst = (kImpHealth + kPistolMin - 1) / kPistolMin;
14     const double average = (kPistolMin + kPistolMax) / 2.0;
15     const double shotsAvg = kImpHealth / average;
16     const double shotgunAvg = average * kPellets;
17
18     constexpr auto kInt = "{:<22}{:>6}\n"; // шаблон для целых
19     constexpr auto kReal = "{:<22}{:>6.1f}\n"; // и для дробных
20     std::cout << std::format(kInt, "Imp health:", kImpHealth);
21     std::cout << std::format(kInt, "Pistol, lucky:", best);
22     std::cout << std::format(kInt, "Pistol, unlucky:", worst);
23     std::cout << std::format(kReal, "Pistol, on average:", shotsAvg);
24     std::cout << std::format(kReal, "Shotgun average:", shotgunAvg);
25 }

```

```

$ ./a.out
Imp health:                60
Pistol, lucky:              4
Pistol, unlucky:           12
Pistol, on average:         6.0
Shotgun average:           70.0

```

Листинг 2.12 — Калькулятор: сколько выстрелов нужно на импа

Строки 5–8 — константы с параметрами оружия и монстра. Никаких магических чисел в расчётах.

Строки 12–13 — трюк, который стоит запомнить. Нам нужно деление с округлением вверх: если до смерти импа не хватает хотя бы единицы урона, нужен ещё один выстрел. Целое деление округляет вниз, поэтому перед делением прибавляем делитель без единицы. При везении: $(60 + 14) / 15 = 74 / 15 = 4$ выстрела. При невезении: $(60 + 4) / 5 = 64 / 5 = 12$. Проверь на бумаге: если бы урона было 61, вышло бы $(61 + 14) / 15 = 5$ — как и должно быть.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, [купив полную легальную версию](#) на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.