

MINI DOOM

Современный C++ через игру



ТОМ 2

Окно в ад

C++20 · Visual Studio 2026

18+

8+

Алексей Примак

**MINI DOOM современный C
++ через игру том 2 окно в ад**

«Автор»

2026

Примак А.

MINI DOOM современный C++ через игру том 2 окно в ад /
А. Примак — «Автор», 2026

Второй том серии «MINI DOOM. Современный C++ через игру». Пора выбираться из консоли: открываем своё окно Windows и рисуем пиксель за пикселем. Ты разберёшься, как устроены оконная процедура и цикл сообщений, что такое кадровый буфер и цвет в одном числе, почему игра не должна зависеть от FPS и как с этим помогает dt. Научишься читать клавиатуру и мышь через Raw Input, отделишь логику игры от Windows стеной, которая позволит тестировать её без окна, и выведешь текст растровым шрифтом, нарисованным прямо в коде. Финал тома — окно, где игрок слушается клавиатуры и мыши, а в углу горит счётчик кадров. Только современный C++20 и WinAPI, строгие предупреждения, настоящие скриншоты. 18+.

Содержание

Вместо предисловия	5
Серия целиком	6
Часть II. Окно в ад	7
Глава 9. Своё окно	8
Как Windows разговаривает с программой	8
Первое окно	9
Сборка	13
Итоги главы	15
Упражнения	15
Задачи	17
Глава 10. Пиксели	19
Пиксель и цвет	19
Картинка в памяти	21
Холст: класс Canvas	24
Конец ознакомительного фрагмента.	25

Алексей Прима́к

MINI DOOM современный C++ через игру том 2 окно в ад

Вместо предисловия

Это второй том книги «MINI DOOM. Современный C++ через игру». В первом томе ты прошёл основы C++ в консоли: переменные, ветвления, функции, контейнеры, свои типы, классы и память — и закончил консольным рогаликом, который проверяет AddressSanitizer.

Теперь из консоли выходим в окно: пиксели, игровой цикл, управление и текст на экране. Номера глав, листингов и рисунков в серии сквозные, поэтому ссылка «см. главу 12» ведёт туда же, куда и в однотомнике.

Серия целиком

— **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. Каждая глава начинается с проекта предыдущей, так что можно начать читать этот том, взяв готовый код последней главы прошлого тома.

Часть II. Окно в ад

Консоль своё отслужила. Чёрный экран с буквами годится для рогалика, но Doom — это пиксели: текстуры, огонь, кровь. В этой части мы откроем настоящее окно Windows, научимся красить в нём каждую точку, заведём игровой цикл, который крутится сотни раз в секунду, подключим клавиатуру и мышь и напишем свой шрифт. К концу части по окну будет бегать квадратный «игрок», которого ты ведёшь мышью и клавишами, а в углу будет тикать счётчик кадров.

Одно важное устройство этой части. Код, который общается с Windows, мы с самого начала будем держать отдельно от кода, который рисует и считает. Так сделан и настоящий MINI DOOM: вся игра не знает о Windows ничего, а маленький файл `main_win32.cpp` передаёт ей нажатия клавиш и показывает готовые кадры. Поэтому код игры можно проверять автотестами без всякого окна — и мы будем это делать.

Лирика. Все скриншоты окон в этой части настоящие: каждую программу я собирал компилятором под Windows со строгими предупреждениями и запускал, а клавиши и мышь нажимал робот по сценарию. Снимал я в Wine — это слой, который запускает Windows-программы на Linux, — поэтому рамка окна на картинках по-старинному голубая, как в Windows 2000. У тебя на Windows 11 рамка будет современной, а всё, что внутри окна, совпадёт пиксель в пиксель.

Глава 9. Своё окно

Окно в Windows — не просто прямоугольник на экране. Это объект, который живёт своей жизнью: его двигают, сворачивают, закрывают, перекрывают другими окнами, и обо всём этом Windows сообщает твоей программе. Программа должна внимательно слушать и вовремя отвечать, иначе Windows решит, что она зависла, и напишет в заголовке «Не отвечает».

В этой главе мы напишем минимальную программу с окном на чистом Win32 API — без библиотек и мастеров проектов. Строк будет немного, но каждая важна: на этом скелете будет висеть весь MINI DOOM.

Как Windows разговаривает с программой

Представь окно ресторана выдачи. Windows — кухня, которая непрерывно подсовывает в окошко записки: «по окну щёлкнули мышью», «нажали клавишу», «окно надо перерисовать», «окно закрывают». Каждая записка называется **сообщением**. Записки складываются в **очередь сообщений** твоей программы, и программа должна по одной вынимать их и обрабатывать.

Вынимает записки **цикл сообщений** — обычный while в конце wWinMain. А обрабатывает их **оконная процедура** — функцию, которую ты пишешь сам и сообщаем Windows её адрес. Windows вызывает её для каждого сообщения, а ты решаешь: ответить самому или передать стандартному обработчику. Вся схема — на рисунке 9.1.

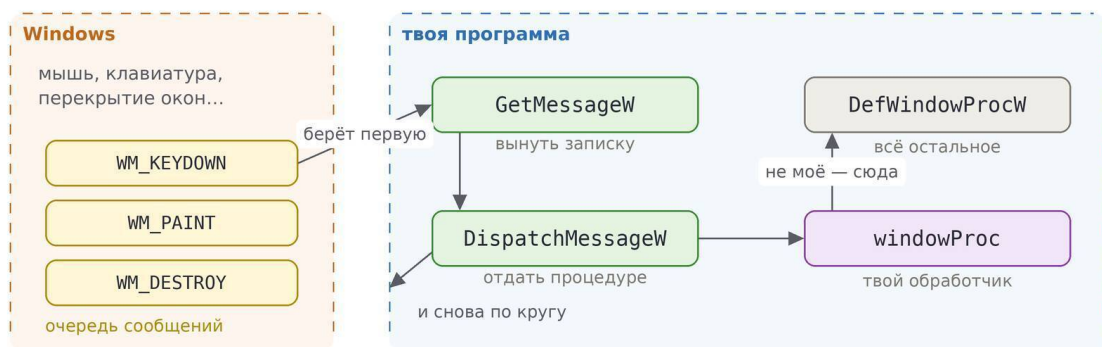
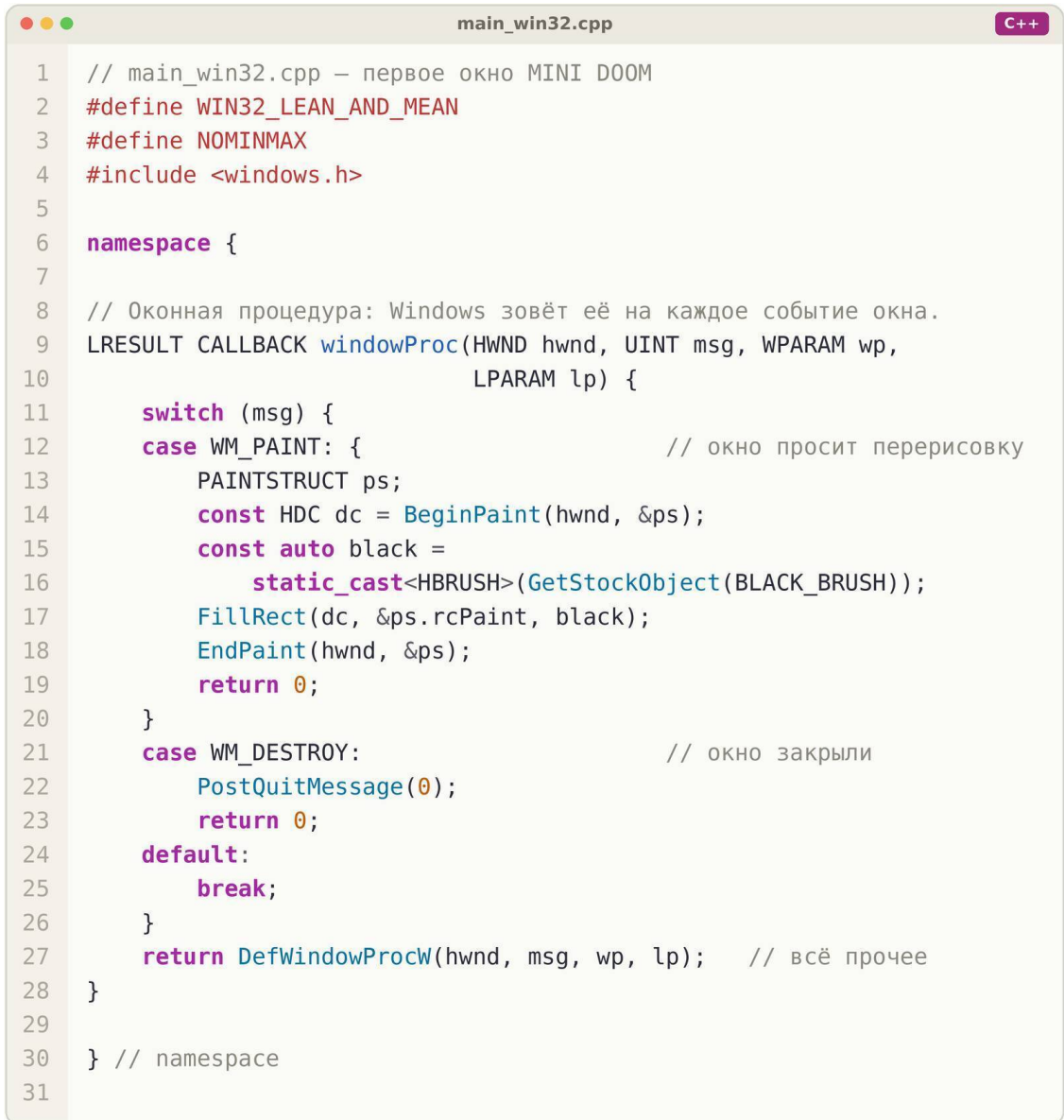


Рисунок 9.1 — Цикл сообщений: Windows кладёт записки в очередь, программа вынимает их и отдаёт оконной процедуре

Запомни. В Windows не программа ходит за событиями, а события приходят к программе. Твоя задача — крутить цикл сообщений и быстро отвечать в оконной процедуре. Застрял в обработчике на пять секунд — окно «не отвечает».

Первое окно

Вот программа целиком — листинг 9.1. Она открывает окно 1280 × 800 с заголовком MINI DOOM и заливает его чёрным. Файл назван `main_win32.cpp`: в нём живёт всё, что связано с Windows, и в следующих главах этот файл будет только расти.



```

1  // main_win32.cpp — первое окно MINI DOOM
2  #define WIN32_LEAN_AND_MEAN
3  #define NOMINMAX
4  #include <windows.h>
5
6  namespace {
7
8  // Оконная процедура: Windows зовёт её на каждое событие окна.
9  LRESULT CALLBACK windowProc(HWND hwnd, UINT msg, WPARAM wp,
10                               LPARAM lp) {
11      switch (msg) {
12      case WM_PAINT: { // окно просит перерисовку
13          PAINTSTRUCT ps;
14          const HDC dc = BeginPaint(hwnd, &ps);
15          const auto black =
16              static_cast<HBRUSH>(GetStockObject(BLACK_BRUSH));
17          FillRect(dc, &ps.rcPaint, black);
18          EndPaint(hwnd, &ps);
19          return 0;
20      }
21      case WM_DESTROY: // окно закрыли
22          PostQuitMessage(0);
23          return 0;
24      default:
25          break;
26      }
27      return DefWindowProcW(hwnd, msg, wp, lp); // всё прочее
28  }
29
30  } // namespace
31

```

Листинг 9.1 — Минимальная программа с окном

```

main_win32.cpp (продолжение) C++
32 int WINAPI wWinMain(_In_ HINSTANCE instance, _In_opt_ HINSTANCE,
33                     _In_ PWSTR, _In_ int show) {
34     WNDCLASSEXW wc{}; // 1. описываем класс окна
35     wc.cbSize = sizeof(wc);
36     wc.lpfnWndProc = windowProc;
37     wc.hInstance = instance;
38     wc.hCursor = LoadCursorW(nullptr, IDC_ARROW);
39     wc.lpszClassName = L"MiniDoomWindow";
40     if (RegisterClassExW(&wc) == 0) {
41         return 1;
42     }
43
44     RECT rect{0, 0, 1280, 800}; // 2. размер картинки
45     AdjustWindowRect(&rect, WS_OVERLAPPEDWINDOW, FALSE);
46     const HWND hwnd = CreateWindowExW(
47         0, wc.lpszClassName, L"MINI DOOM", WS_OVERLAPPEDWINDOW,
48         CW_USEDEFAULT, CW_USEDEFAULT, rect.right - rect.left,
49         rect.bottom - rect.top, nullptr, nullptr, instance, nullptr);
50     if (hwnd == nullptr) {
51         return 1;
52     }
53     ShowWindow(hwnd, show);
54
55     MSG msg{}; // 3. цикл сообщений
56     while (GetMessageW(&msg, nullptr, 0, 0) > 0) {
57         TranslateMessage(&msg);
58         DispatchMessageW(&msg);
59     }
60     return static_cast<int>(msg.wParam);
61 }

```

Листинг 9.1 (продолжение) — Минимальная программа с окном

Собираем, запускаем — и вот оно, рисунок 9.2. Пустое чёрное окно, но это **наше** окно: его можно двигать, сворачивать, разворачивать на весь экран и закрывать крестиком.

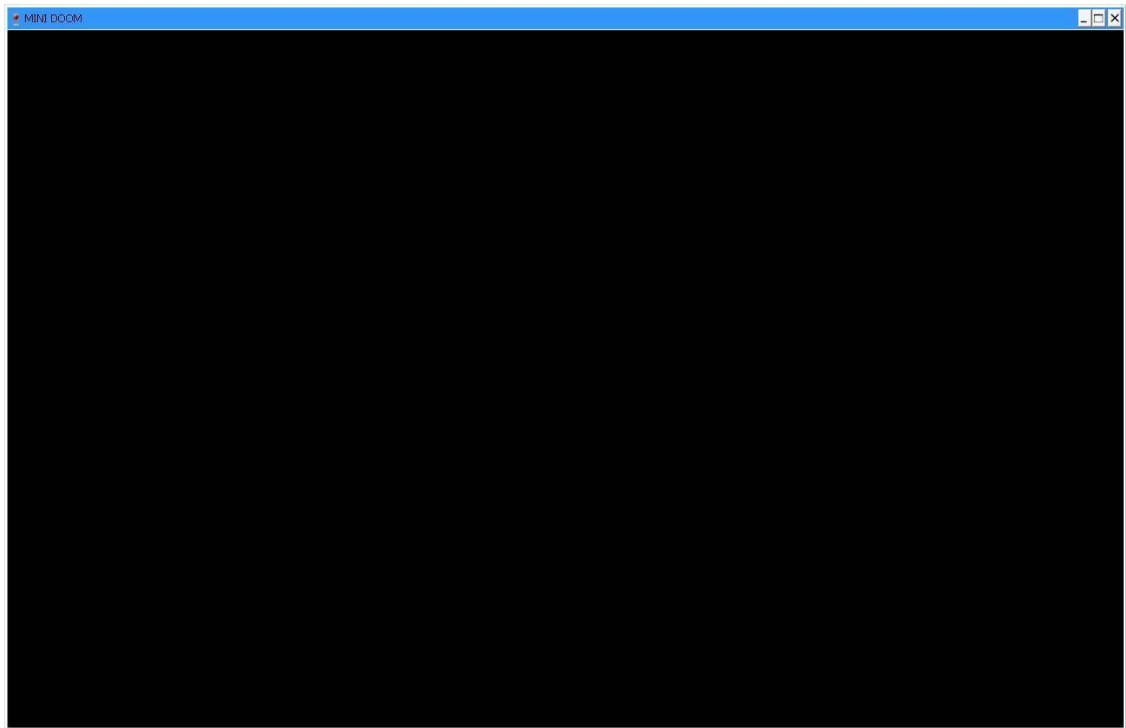


Рисунок 9.2 — Первое окно MINI DOOM — настоящий запуск программы из листинга 9.1

Разберём три главных шага, они отмечены в комментариях цифрами.

Шаг 1, строки 34–42 — класс окна. Прежде чем создать окно, Windows хочет знать его «породу»: какая у окон этого вида оконная процедура, какой курсор, как называется порода. Это описание — структура `WNDCLASSEXW` — регистрируется функцией `RegisterClassExW`. Пустые фигурные скобки в объявлении `ws{ }` обнуляют все поля, а мы заполняем только нужные. Поле `cbSize` — размер самой структуры: так Windows понимает, с какой версией структуры имеет дело.

Шаг 2, строки 44–53 — само окно. Мы хотим, чтобы картинка внутри окна была ровно 1280×800 , но окно — это ещё рамка и заголовок. `AdjustWindowRect` берёт желаемый размер картинки и увеличивает его на толщину рамки и заголовка. Потом `CreateWindowExW` создаёт окно по зарегистрированному классу, а `ShowWindow` показывает его. Чем окно отличается от его **клиентской области**, видно на рисунке 9.3.

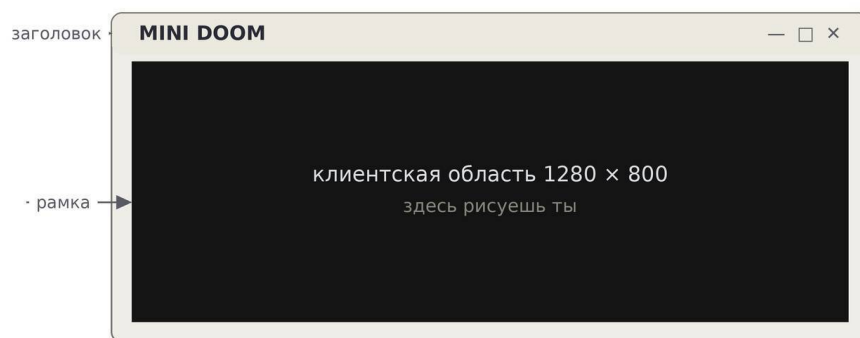


Рисунок 9.3 — Окно — это заголовок, рамка и клиентская область; `AdjustWindowRect` добавляет рамку к нужному размеру картинки

Шаг 3, строки 55–59 — цикл сообщений. GetMessageW ждёт, пока в очереди появится записка, и кладёт её в msg. TranslateMessage превращает нажатия клавиш в символы — для игры это не важно, но так принято. DispatchMessageW отдаёт записку нашей оконной процедуре. Цикл крутится, пока GetMessageW возвращает положительное число. Когда приходит сообщение о выходе, она вернёт ноль — и программа завершится.

Строки 9–28 — оконная процедура. Мы обрабатываем всего два сообщения. WM_PAINT приходит, когда окну нужно перерисоваться — например, его вытащили из-за другого окна. Мы заливаем всё чёрным между BeginPaint и EndPaint. WM_DESTROY приходит, когда окно уничтожено, и PostQuitMessage(0) кладёт в очередь то самое сообщение о выходе, которое остановит цикл. Всё остальное — передвижение, изменение размера, клик по крестик — отдаём стандартному обработчику DefWindowProcW на строке 27.

Обрати внимание на букву W на конце почти всех функций и на букву L перед строками: L"MINI DOOM". Это «широкие» версии функций и строк для Unicode. Обычная строка хранит символы в char, по байту на символ, а широкая — в wchar_t. Сколько места это занимает в Windows, покажет листинг 9.2: я собрал его компилятором под Windows и запустил как настоящую Windows-программу.

```

1  #include <iostream>
2  #include <string>
3
4  int main() {
5      const std::string narrow = "DOOM";
6      const std::wstring wide = L"DOOM";
7      std::cout << "char:      " << sizeof(char) << " byte\n";
8      std::cout << "wchar_t:  " << sizeof(wchar_t) << " bytes\n";
9      std::cout << "\"DOOM\":   " << narrow.size() * sizeof(char)
10         << " bytes\n";
11      std::cout << "L\"DOOM\":  " << wide.size() * sizeof(wchar_t)
12         << " bytes\n";
13      std::cout << "long:     " << sizeof(long) << " bytes\n";
14  }

```

```

$ main.exe # собрано под Windows
char:      1 byte
wchar_t: 2 bytes
"DOOM":   4 bytes
L"DOOM":  8 bytes
long:     4 bytes

```

Листинг 9.2 — Размеры обычных и широких строк в Windows

В Windows wchar_t занимает два байта, и строка L"DOOM" весит вдвое больше обычной. Этими двухбайтовыми символами — кодировкой UTF-16 — Windows и говорит с программами начиная с Windows NT. Поэтому мы пишем W-функции и L-строки: так русские, японские и любые другие буквы в заголовках окон работают без кракозябр.

А последняя строка вывода — привет из главы 2. Помнишь грабли про long, который в Windows занимает 4 байта, а в Linux — 8? Вот они, живём: Windows-программа честно

сообщает 4. Если собрать тот же файл под Linux, получится `wchar_t` в 4 байта и `long` в 8. Типы в C++ не обязаны иметь одинаковый размер на разных системах, и об этом полезно помнить.

Грабли. Не путай `sizeof` и длину строки. `sizeof(L"DOOM")` — это размер массива символов вместе с невидимым нулём в конце, а не число букв. Для длины строки бери `std::wstring` и его `size()`.

Функция	Что делает
<code>RegisterClassExW</code>	регистрирует «породу» окон: процедуру, курсор, имя
<code>AdjustWindowRect</code>	добавляет к размеру картинки рамку и заголовок
<code>CreateWindowExW</code>	создаёт окно указанной породы
<code>ShowWindow</code>	показывает окно на экране
<code>GetMessageW</code>	ждёт и вынимает сообщение из очереди
<code>DispatchMessageW</code>	передаёт сообщение оконной процедуре
<code>DefWindowProcW</code>	стандартная обработка всех сообщений, которые не нужны нам
<code>PostQuitMessage</code>	кладёт в очередь сообщение о выходе

Таблица 9.1 — Функции Win32, из которых собрано первое окно

Сборка

В Visual Studio создай **пустой проект C++** — не «Консольное приложение» и не «Классическое приложение Windows», последнее генерирует кучу лишнего кода. Добавь файл `main_win32.cpp`, выставь настройки строгости из таблицы 1.2 и ещё две из таблицы 9.2.

Где найти	Что поставить	Зачем
Компоновщик → Система → Подсистема	Windows (<code>/SUBSYSTEM:WINDOWS</code>)	без чёрной консоли; точка входа — <code>wWinMain</code>
Дополнительно → Набор символов	Использовать набор символов Юникода	W-функции и L-строки; обычно уже стоит

Таблица 9.2 — Дополнительные настройки проекта с окном

Я собираю тот же файл компилятором `clang` под Windows — листинг 9.3. Первая попытка специально без режима Unicode, вторая — правильная.

```

# без режима Unicode
$ x86_64-w64-mingw32-clang++ -std=c++20 -Wall -Wextra -Werror
  main_win32.cpp -o doom.exe
main_win32.cpp:38:18: error: no matching function for call to
  'LoadCursorW'
    38 |         wc.hCursor = LoadCursorW(nullptr, IDC_ARROW);
        |                        ^~~~~~
winuser.h:4116:29: note: candidate function not viable: no known
  conversion from 'LPSTR' (aka 'char *') to 'LPCWSTR' (aka 'const
  wchar_t *') for 2nd argument
    4116 |     WINUSERAPI HCURSOR WINAPI LoadCursorW(HINSTANCE
        |                                           ^
        ~~~~~~
1 error generated.
# ... отчёт сокращён до сути
# с Unicode и подсистемой Windows
$ x86_64-w64-mingw32-clang++ -std=c++20 -Wall -Wextra -Werror -municode
  -mwindows main_win32.cpp -o doom.exe
$ file doom.exe
doom.exe: PE32+ executable (GUI) x86-64, for MS Windows, 13 sections

```

Листинг 9.3 — Сборка под Windows: без Unicode и с ним

Без Unicode сборка падает на строке 38. Дело в том, что у большинства функций Windows по две версии: LoadCursorA для старых однобайтовых строк и LoadCursorW для Unicode. Константа IDC_ARROW тоже бывает двух видов, и какой именно — решает макрос UNICODE. Мы вызвали W-функцию, а IDC_ARROW оказался A-вида — типы не сошлись. Флаг -municode включает Unicode и заодно говорит, что точка входа — wWinMain. Флаг -mwindows — что это оконная программа, а не консольная. Во второй раз компилятор промолчал, а команда file подтвердила: получился исполняемый файл Windows с графическим интерфейсом, GUI.

Грабли. Если создать в Visual Studio «Консольное приложение» и написать в нём wWinMain, компоновщик выдаст ошибку LNK2019: неразрешённый внешний символ main. Консольная программа ищет точку входа main, оконная — WinMain или wWinMain. Лечение — поменять подсистему на Windows по таблице выше, и всё соберётся.

Лирика. Функции, которыми мы пользуемся, появились в Windows NT 3.1 в 1993 году — в тот же год, когда вышел Doom. Сам Doom тогда работал под DOS и рисовал прямо в видеопамять, а под Windows его перенесли только через три года. Интерфейс Win32 с тех пор почти не менялся: программа из этой главы собралась бы и тридцать лет назад, и запустится на Windows 11.

По-взрослому. Настоящие игры редко пишут на голом Win32. Обычно берут библиотеку вроде SDL или GLFW: она прячет окно, ввод и звук за одинаковым интерфейсом для Windows, Linux и macOS. Мы идём по сложному пути специально — чтобы понять, что такие библиотеки делают внутри. А благодаря отделению игры от платформы перенести MINI DOOM на SDL потом можно будет, переписав один файл.

Итоги главы

- Windows общается с программой сообщениями, которые копятся в очереди.
- Цикл сообщений вынимает их GetMessageW и отдаёт DispatchMessageW оконной процедуре.
- Оконная процедура обрабатывает нужные сообщения, остальные отдаёт DefWindowProcW.
- Окно создаётся по зарегистрированному классу; AdjustWindowRect считает размер с рамкой.
- WM_DESTROY → PostQuitMessage → GetMessageW возвращает 0 → программа выходит.
- Функции и строки — W-версии и L"..."; подсистема — Windows, точка входа — wWinMain.

Контрольная точка: у тебя есть чёрное окно MINI DOOM, которое честно отвечает Windows и закрывается крестиком. В главе 10 мы перестанем заливать его чёрной кистью и начнём красить каждый пиксель сами.

Упражнения

Упражнение 9.1. *Что выведет?* Программа из листинга 9.4 собрана под Windows. Что она напечатает? Будь внимателен с последним числом.

```

main.cpp
1  #include <iostream>
2  #include <string>
3
4  int main() {
5      const std::wstring title = L"MINI DOOM";
6      std::cout << title.size() << ' '
7                << title.size() * sizeof(wchar_t) << ' '
8                << sizeof(L"MINI DOOM") << '\n';
9  }

```

Листинг 9.4 — Упражнение 9.1

Упражнение 9.2. Найди баг. В программе из листинга 9.5 окно открывается и закрывается крестиком, но если посмотреть в диспетчер задач, процесс doom.exe остаётся висеть. Что сломано?

```

main_win32.cpp
1  // main_win32.cpp — первое окно MINI DOOM
2  #define WIN32_LEAN_AND_MEAN
3  #define NOMINMAX
4  #include <windows.h>
5
6  namespace {
7
8  // Оконная процедура: Windows зовёт её на каждое событие окна.
9  LRESULT CALLBACK windowProc(HWND hwnd, UINT msg, WPARAM wp,
10                             LPARAM lp) {
11      switch (msg) {
12          case WM_PAINT: { // окно просит перерисовку
13              PAINTSTRUCT ps;
14              const HDC dc = BeginPaint(hwnd, &ps);
15              const auto black =
16                  static_cast<HBRUSH>(GetStockObject(BLACK_BRUSH));
17              FillRect(dc, &ps.rcPaint, black);
18              EndPaint(hwnd, &ps);
19              return 0;
20          }
21          case WM_DESTROY: // окно закрыли
22              return 0;
23          default:
24              break;
25      }
26      return DefWindowProcW(hwnd, msg, wp, lp); // всё прочее
27  }
28
29  } // namespace
30

```

Листинг 9.5 — Упражнение 9.2

```

main_win32.cpp (продолжение)
31 int WINAPI wWinMain(_In_ HINSTANCE instance, _In_opt_ HINSTANCE,
32                     _In_ PWSTR, _In_ int show) {
33     WNDCLASSEXW wc{}; // 1. описываем класс окна
34     wc.cbSize = sizeof(wc);
35     wc.lpfnWndProc = windowProc;
36     wc.hInstance = instance;
37     wc.hCursor = LoadCursorW(nullptr, IDC_ARROW);
38     wc.lpszClassName = L"MiniDoomWindow";
39     if (RegisterClassExW(&wc) == 0) {
40         return 1;
41     }
42
43     RECT rect{0, 0, 1280, 800}; // 2. размер картинки
44     AdjustWindowRect(&rect, WS_OVERLAPPEDWINDOW, FALSE);
45     const HWND hwnd = CreateWindowExW(
46         0, wc.lpszClassName, L"MINI DOOM", WS_OVERLAPPEDWINDOW,
47         CW_USEDEFAULT, CW_USEDEFAULT, rect.right - rect.left,
48         rect.bottom - rect.top, nullptr, nullptr, instance, nullptr);
49     if (hwnd == nullptr) {
50         return 1;
51     }
52     ShowWindow(hwnd, show);
53
54     MSG msg{}; // 3. цикл сообщений
55     while (GetMessageW(&msg, nullptr, 0, 0) > 0) {
56         TranslateMessage(&msg);
57         DispatchMessageW(&msg);
58     }
59     return static_cast<int>(msg.wParam);
60 }

```

Листинг 9.5 (продолжение) — Упражнение 9.2

Упражнение 9.3. *Напиши сам.* Сделай так, чтобы окно закрывалось по клавише Esc. Подсказка: нажатие клавиши приходит сообщением WM_KEYDOWN, а код клавиши лежит в параметре wp; у Esc это VK_ESCAPE.

Упражнение 9.4. *Подумай.* Что случится, если в обработчике WM_PAINT вместо пары BeginPaint и EndPaint просто ничего не делать и вернуть 0?

Задачи

Задача 9.1. Окно фиксированного размера. Сделай окно, у которого нельзя менять размер мышью и которое нельзя развернуть на весь экран. Подсказка: стиль WS_OVERLAPPEDWINDOW — это комбинация нескольких флагов; убери из неё лишние оператором & и ~.

Задача 9.2. Подтверждение выхода. При нажатии крестика Windows присылает WM_CLOSE. Перехвати его и спроси игрока через MessageBoxW, точно ли он хочет выйти. Закрой окно, только если ответ «Да».

Задача 9.3. Счётчик сообщений. Заведи в оконной процедуре статический счётчик и выводи число полученных сообщений в заголовок окна через SetWindowTextW. Подвигай окно мышью и посмотри, сколько сообщений приходит за секунду.

Глава 10. Пиксели

Чёрное окно из прошлой главы залито кистью Windows. Это удобно, но для игры бесполезно: кисть умеет красить прямоугольники, а нам нужны огонь, текстуры и рожа импа. Поэтому мы поступим, как настоящие игры девяностых: заведём в памяти свой кусок экрана, будем красить в нём каждую точку сами, а Windows попросим только показать готовую картинку.

В этой главе разберёмся, как устроен цвет, как двумерная картинка лежит в одномерной памяти и как одним вызовом растянуть её на всё окно. Попутно наступим на три настоящих грабли — и все три снимем на скриншоты.

Пиксель и цвет

Экран — это сетка крошечных лампочек, **пикселей**. Каждый пиксель светится смесью трёх цветов: красного, зелёного и синего — отсюда сокращение RGB. Яркость каждого канала — число от 0 до 255, ровно один байт. Все три по нулям — чёрный, все по 255 — белый, красный 255 при нулевых остальных — чистый красный. Всего так можно смешать 256^3 , то есть почти 17 миллионов цветов.

Три байта удобно хранить в одном 32-битном числе `std::uint32_t`. Мы будем раскладывать их так: старший байт пустой, дальше красный, зелёный и синий. В шестнадцатеричной записи это выглядит как `0x00RRGGBB` — каждые две цифры ровно байт, поэтому цвет читается прямо глазами. Как собрать такое число из трёх каналов и разобрать обратно, показывает листинг 10.1.

```

main.cpp C++
1  #include <cstdint>
2  #include <format>
3  #include <iostream>
4
5  // Упаковать три канала в одно число 0x00RRGGBB.
6  constexpr std::uint32_t rgb(std::uint32_t r, std::uint32_t g,
7                               std::uint32_t b) {
8      return (r << 16) | (g << 8) | b;
9  }
10
11 int main() {
12     const std::uint32_t fire = rgb(255, 120, 40); // огонь
13     std::cout << std::format("fire   = 0x{:08X}\n", fire);
14
15     // Распаковка: сдвигаем нужный байт вниз и отрезаем маской.
16     const std::uint32_t r = (fire >> 16) & 0xFF;
17     const std::uint32_t g = (fire >> 8) & 0xFF;
18     const std::uint32_t b = fire & 0xFF;
19     std::cout << std::format("r={} g={} b={}\n", r, g, b);
20
21     // Затемнить вдвое: каждый канал пополам.
22     const std::uint32_t dark = rgb(r / 2, g / 2, b / 2);
23     std::cout << std::format("dark   = 0x{:08X}\n", dark);
24     const std::uint32_t white = rgb(255, 255, 255);
25     std::cout << std::format("white  = 0x{:08X}\n", white);
26 }

```

```

$ ./a.out
fire   = 0x00FF7828
r=255 g=120 b=40
dark   = 0x007F3C14
white  = 0x00FFFFFF

```

Листинг 10.1 — Упаковка и распаковка цвета сдвигами и масками

Строка 8 собирает цвет. Оператор << сдвигает биты числа влево: $r \ll 16$ переносит красный на 16 бит выше, в третий байт; $g \ll 8$ — во второй. Синий остаётся на месте. Оператор | («или» для битов) складывает три сдвинутых числа в одно, не мешая друг другу, — ведь каждое занимает свой байт. Как биты разъезжаются по местам, показано на рисунке 10.1.

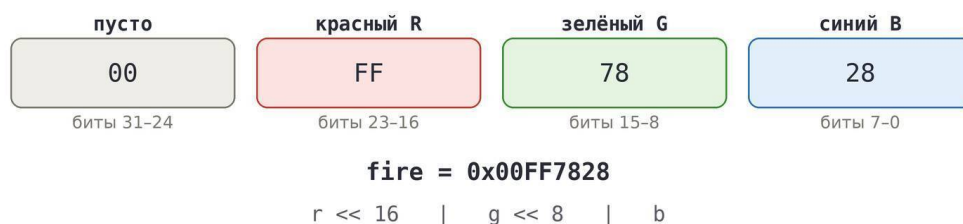


Рисунок 10.1 — Цвет в одном 32-битном числе: по байту на канал

Строки 16–18 разбирают цвет обратно. Сдвиг вправо опускает нужный байт в самый низ, а **маска** `& 0xFF` отрезает всё, что выше: оставляет только младшие восемь бит. В выводе видно, что из `0x00FF7828` вернулись те же 255, 120 и 40.

Строка 22 затемняет цвет вдвое — каждый канал пополам. Этим приёмом в главе 18 мы будем погружать дальние стены во тьму. А формат `{:08X}` в `std::format` печатает число шестнадцатеричными цифрами, восемь знаков с ведущими нулями: так байты цвета видны по отдельности.

Запомни. `x << n` — сдвиг влево на `n` бит, `x >> n` — вправо, `x & mask` — оставить только биты маски, `a | b` — сложить биты. Четыре операции, на которых держится вся работа с цветом.

Картинка в памяти

Наша картинка — 640×400 пикселей. Как хранить двумерную сетку? Можно вектором векторов, но это четыреста отдельных кусков памяти. Игры поступают проще: один длинный ряд из $640 \times 400 = 256\,000$ чисел, строки картинки лежат друг за другом, как вагоны поезда. Пиксель (x, y) живёт по номеру $y \times \text{width} + x$: пропусти y целых строк и отсчитай x в нужной. Проверим на крошечном экране 4×3 — листинг 10.2.

```

main.cpp C++
1  #include <format>
2  #include <iostream>
3  #include <vector>
4
5  int main() {
6      const int width = 4;
7      const int height = 3;
8      std::vector<char> screen(width * height, '.');
9
10     // Пиксель (x, y) лежит в векторе под номером y * width + x.
11     auto at = [&](int x, int y) -> char& {
12         return screen[static_cast<std::size_t>(y * width + x)];
13     };
14     at(0, 0) = 'A';
15     at(3, 0) = 'B';
16     at(1, 2) = 'C';
17
18     for (char c : screen) { // как лежит в памяти
19         std::cout << c;
20     }
21     std::cout << '\n';
22     for (int y = 0; y < height; ++y) { // как видим на экране
23         for (int x = 0; x < width; ++x) {
24             std::cout << at(x, y);
25         }
26         std::cout << '\n';
27     }
28     std::cout << std::format("C is at index {}\n", 2 * width + 1);
29 }

```

```

$ ./a.out
A..B.....C..
A..B
....
.C..
C is at index 9

```

Листинг 10.2 — Двумерный экран в одномерном векторе

Первая строка вывода — память как она есть: двенадцать символов подряд. Остальные три — тот же вектор, нарезанный по четыре: так его видим мы. Буква С в точке (1, 2) лежит под номером $2 \times 4 + 1 = 9$. Схема — на рисунке 10.2.

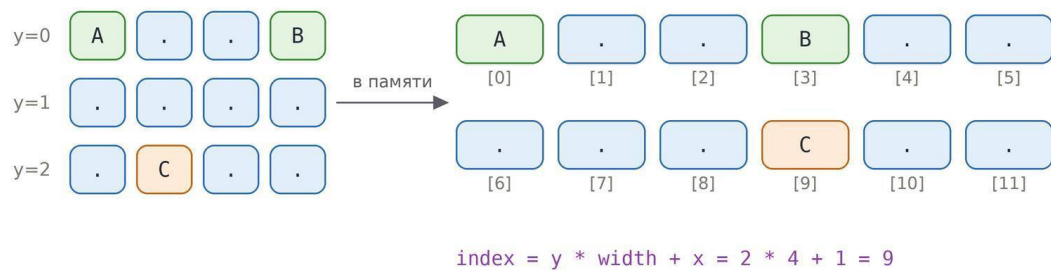


Рисунок 10.2 — Экран 4×3 и тот же экран в памяти: строки идут друг за другом

У такой укладки есть ловушка, и она в листинге 10.3. Программа хочет обвести экран 640×400 рамкой: левый край — пиксель $x = 0$, правый — $x = \text{width}$. Собрано с санитайзером.

```

main.cpp
1  #include <cstdint>
2  #include <vector>
3
4  int main() {
5      const int width = 640;
6      const int height = 400;
7      std::vector<std::uint32_t> pixels(width * height);
8
9      // Рамка вокруг экрана: правый край —  $x = \text{width}$ ... или нет?
10     for (int y = 0; y < height; ++y) {
11         const auto row = static_cast<std::size_t>(y * width);
12         pixels[row] = 0xFFFFFFFF; // левый
13         pixels[row + static_cast<std::size_t>(width)] = 0xFFFFFFFF;
14     }
15     return 0;
16 }

```

```

$ ./a.out # собрано с -fsanitize=address,undefined
ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7fbee09ff800
WRITE of size 4 at 0x7fbee09ff800 thread T0
    main main.cpp:13
0x7fbee09ff800 is located 0 bytes after 1024000-byte region
[0x7fbee0905800,0x7fbee09ff800)
allocated by thread T0 here:
    main main.cpp:7
SUMMARY: AddressSanitizer: heap-buffer-overflow main.cpp:13 in main
# ... отчёт сокращён до сути

```

Листинг 10.3 — Правый край на пиксель дальше, чем нужно

Санитайзер нашёл запись за концом вектора на строке 13. Но посмотри внимательнее, **когда** он её нашёл: на последней, четырёхсотой строке. Правый край — $x = \text{width}$, а последний пиксель строки — $\text{width} - 1$. Для всех строк, кроме последней, пиксель « $x = \text{width}$ » — это

на самом деле первый пиксель **следующей** строки. Программа молча красила левый край строкой ниже, и ни один сторож этого не заметил. Только на последней строке «следующей строки» уже нет, и запись ушла за конец памяти.

Грабли. Ошибка на единицу в пиксельной графике коварна вдвойне: в одномерной памяти выход за край строки — это не выход за край массива, а заход в соседнюю строку. Санитайзер такое не ловит. Поэтому все обращения к пикселям мы будем делать через одну функцию, которая проверяет обе координаты, — тогда ошибиться можно только в одном месте.

Холст: класс Canvas

Обернём кусок памяти в класс — как учились в главе 7. Холст знает свою ширину и высоту, умеет залить всё одним цветом, поставить точку и закрасить прямоугольник. И ничего не знает про Windows: это обычный C++. Заголовок — листинг 10.4.

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.