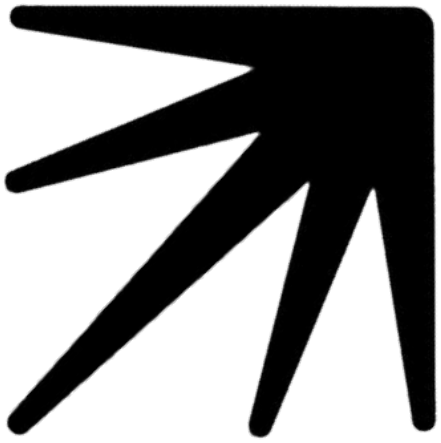
Студенческое конструкторское бюро Поток



Методические указания
к выполнению лабораторной работы № 0

# C/C++
# и устройство программы

Автор: Беляев А. А.

МГТУ им. Н.Э. Баумана, г. Москва
2026

# Содержание

**Введение** **4**
1. Назначение ЛР0 . . . 4
2. Как устроена лабораторная работа . . . 4
3. Содержание работы . . . 5
4. Ожидаемый результат . . . 5

## Теоретическая часть 7

**Глава 1. Основы C/C++ и базовые конструкции языка** **7**
1.1. Исходный текст программы: инструкции, блоки, комментарии . . . 7
1.2. Переменные, инициализация и основные типы данных . . . 9
1.3. Выражения, операторы и преобразования типов . . . 14
1.4. Условия и циклы . . . 20
1.5. Функции: объявление, определение, параметры и возвращаемое значение . . . 29
1.6. Ввод и вывод в консоль . . . 35
1.7. Контрольные вопросы . . . 41

**Глава 2. Работа с данными и составными типами** **43**
2.1. Адрес объекта и указатели . . . 43
2.2. Ссылки в C++ . . . 45
2.3. Массивы: элементы, границы и передача в функции . . . 47
2.4. Структуры и передача составных данных . . . 50
2.5. Перечисления . . . 54
2.6. Двоичное представление целых чисел и побитовые операции . . . 56
2.7. Контрольные вопросы . . . 59

**Глава 3. Объекты, время жизни и организация памяти** **61**
3.1. Область видимости, время жизни и длительность хранения . . . 61
3.2. Ограничение изменения объектов: const . . . 64
3.3. Константные выражения и constexpr в C++ . . . 66
3.4. `static` и `extern` в разных контекстах . . . 68
3.5. `volatile`: назначение и ограничения . . . 71
3.6. Стек, статическая память и куча . . . 73
3.7. Контрольные вопросы . . . 75

**Глава 4. Многофайловая программа** **77**
4.1. Разделение программы на модули: интерфейс и реализация . . . 77
4.2. Объявления и определения . . . 77
4.3. Заголовочные файлы и включение через #include . . . 78
4.4. Защита заголовков от повторного включения . . . 79
4.5. Реализация и связь имён между единицами трансляции . . . 79
4.6. Использование модуля и сборка проекта . . . 80
4.7. Контрольные вопросы . . . 81

**Глава 5. Компиляция, объектные файлы и компоновка** **83**
5.1. От исходного текста к исполняемой программе . . . 83
5.2. Препроцессирование . . . 83
5.3. Компиляция и ассемблирование . . . 84

5.4. Символы и компоновка . . . 84
5.5. Ошибки разных этапов . . . 86
5.6. Статические библиотеки . . . 86
5.7. Оптимизация и целевая платформа . . . 87
5.8. Контрольные вопросы . . . 87

**Глава 6. Устройство готовой программы** **89**
6.1. ELF и содержимое готового файла . . . 89
6.2. Секции: код, данные и служебная информация . . . 89
6.3. Секции и сегменты . . . 90
6.4. Адреса выполнения и загрузки: VMA и LMA . . . 91
6.5. Сценарий компоновки и карта памяти . . . 91
6.6. Символы, команды процессора и расход памяти . . . 92
6.7. Map-файл и исключение неиспользуемых секций . . . 92
6.8. BIN и HEX . . . 93
6.9. Контрольные вопросы . . . 93

**Глава 7. Запуск и выполнение программы** **94**
7.1. Состав программы микроконтроллера . . . 94
7.2. Сборка для Cortex-M4 . . . 95
7.3. Сброс и таблица векторов . . . 96
7.4. Подготовка памяти и переход в main . . . 97
7.5. Исходный код и машинные инструкции . . . 98
7.6. Запись программы и наблюдение результата . . . 99
7.7. Что проверять при переносе и ошибках запуска . . . 100
7.8. Контрольные вопросы . . . 101

**Дополнительный раздел. ООП и структуры данных в C++** **102**
8.1. Классы, объекты, методы и пространства имён . . . 102
8.2. Инкапсуляция и доступ к членам класса . . . 104
8.3. Конструкторы, деструкторы и время жизни . . . 106
8.4. Наследование и полиморфизм . . . 111
8.5. Пример: один анализатор и разные правила тревоги . . . 114
8.6. Перегрузка функций . . . 116
8.7. Шаблоны функций и классов . . . 117
8.8. Стек . . . 119
8.9. Очередь и кольцевой буфер . . . 121
8.10. Стандартные контейнеры C++ . . . 122
8.11. Контрольные вопросы . . . 125

**Практическая часть** **128**

**Глава 1** **128**
Блок 1.1. Целочисленные вычисления . . . 128
Блок 1.2. Деление, остаток и разряды числа . . . 129
Блок 1.3. Вещественные вычисления . . . 132
Блок 1.4. Условия и выбор . . . 133
Блок 1.5. Циклы и обработка последовательностей . . . 137
Блок 1.6. Повторение действий и обработка цифр . . . 138
Блок 1.7. Вложенные циклы . . . 140

Блок 1.8. Функции . . . . . 142

**Глава 2** **144**
Блок 2.1. Массивы . . . . . 144
Блок 2.2. Указатели и ссылки . . . . . 152
Блок 2.3. Структуры . . . . . 155
Блок 2.4. Перечисления . . . . . 159
Блок 2.5. Побитовые операции . . . . . 161

**Глава 3** **164**
Блок 3.1. Динамическая память . . . . . 164

**Глава 8. ООП и структуры данных в C++** **168**
Блок 8.1. Геттеры, сеттеры и инкапсуляция . . . . . 168
Блок 8.2. Динамическая память, конструкторы и деструкторы . . . . . 169
Блок 8.3. Наследование и полиморфизм . . . . . 171
Блок 8.4. Перегрузки, шаблоны и пространства имён . . . . . 173
Блок 8.5. Стек . . . . . 175
Блок 8.6. Очередь и кольцевой буфер . . . . . 176
Блок 8.7. Стандартные контейнеры . . . . . 178

**Источники** **181**

# Введение

Версия: 1.0
Дата: 09.10.2026

## 1. Назначение ЛР0

Лабораторная работа №0 предназначена для выравнивания начальной подготовки студентов в области программирования на C и C++ перед переходом к основной части курса. Подготовка может начинаться с нулевого опыта программирования.

Работа формирует общую основу для последующего изучения микроконтроллеров и разработки встроенных программ. Основная задача работы — научить студента писать небольшие программы и понимать код на C и C++. Вопросы сборки, размещения в памяти и запуска программы рассматриваются как необходимое введение в специфику программирования микроконтроллеров.

Практические задания ЛР0 выполняются на компьютере; решения необходимо отправлять на сайте SDB Lab. Подготавливать и отлаживать программы можно в выбранной среде разработки или на сайте. Работа с микроконтроллером, подключение платы и запись прошивки в этой лабораторной работе не предусмотрены. Материал о микроконтроллерах служит теоретической подготовкой к последующим работам курса.

## 2. Как устроена лабораторная работа

Лабораторная работа состоит из теоретической и практической частей. Теоретическая часть разделена на главы, каждая из которых объединяет связанные темы и завершается контрольными вопросами для самостоятельной проверки понимания. Объяснения сопровождаются примерами кода. Практическая часть содержит базовые и дополнительные задания.

Начинающему студенту рекомендуется изучать материал последовательно. Студент, уже знакомый с отдельными темами, может пропускать известные объяснения и возвращаться к соответствующим разделам по мере необходимости.

Обязательное ядро работы составляют главы 1–3 и практика по их темам. Ожидается полное освоение главы 1 и хорошее освоение глав 2–3. Контрольные вопросы этих глав используются для проверки понимания. Главы 4–7 изучаются по возможности: для их освоения полезен опыт программирования, поэтому затруднения при первом чтении ожидаемы. Студент отвечает на контрольные вопросы по мере изучения соответствующего материала. Дополнительный раздел об ООП и структурах данных рекомендуется для углубления понимания работы с памятью и организации данных. Этот раздел и контрольные вопросы к нему не входят в обязательный объём ЛР0.

Теория и практика представлены отдельными частями методички. При прохождении работы рекомендуется переходить к практическим заданиям по мере освоения соответствующих тем. Дополнительные задания служат для закрепления и углубления материала и выполняются по потребности.

Методические указания задают содержание и последовательность изучения и содержат обязательную информацию для выполнения работы. Материал излагается преимущественно в точных и формальных технических формулировках. Методичка не предполагает, что начинающий студент сможет понять каждое объяснение при первом самостоятельном прочтении.

ИИ является предусмотренной частью процесса обучения и выполняет роль индивидуального преподавателя. При появлении непонятного термина, формулировки или примера рекомендуется обращаться к ИИ за пояснением, разбором по шагам и дополнительными примерами. От студента не требуется самостоятельно преодолевать все затруднения при изучении теории только по тексту методички. Практические решения студент разрабатывает самостоятельно; ИИ может объяснять материал и помогать разбирать ошибки. Студент должен понимать собственное решение и уметь объяснить его работу.

## 3. Содержание работы

В работе рассматриваются основные конструкции C, функции, указатели, массивы, структуры, перечисления и побитовые операции. Затем вводятся область видимости и время жизни объектов, способы организации памяти и отдельные возможности C++. Основой изложения является C; средства, относящиеся к C++, обозначаются отдельно и рассматриваются после необходимой языковой базы. Дополнительный раздел посвящён ООП и структурам данных в C++: классам и объектам, инкапсуляции, конструкторам и деструкторам, владению памятью, наследованию и полиморфизму, перегрузкам и шаблонам, а также стеку, очереди, кольцевому буферу и стандартным контейнерам.

Далее изучаются устройство многофайловой программы, назначение заголовочных файлов, отдельная компиляция и компоновка. Рассматриваются структура готового образа, секции программы, форматы ELF, BIN и HEX, а также использование map-файла. Завершающая часть посвящена подготовке программы к выполнению и передаче управления в main.

Примеры языка C соответствуют C11, примеры C++ — C++17. Студент самостоятельно выбирает среду разработки и компилятор, поддерживающий указанные стандарты. Приведённые команды сборки показывают применение GCC и G++; выбор этих инструментов не является обязательным. Для последующих работ с STM32 можно использовать среду производителя, например STM32CubeIDE, объединяющую средства разработки и отладки [2]. Настройка сторонней среды для микроконтроллера может потребовать дополнительных действий и опыта; освоение такой настройки не является требованием ЛР0. Средства прошивки и отладки для последующих работ также выбираются студентом.

Для последующих лабораторных работ курса предполагаются плата Black Pill на STM32F411CEU6 и программатор-отладчик ST-Link V2. Использование другой платы требует адаптации проекта с учётом документации выбранного микроконтроллера и платы. Дополнительное оборудование определяется заданием конкретной лабораторной работы. Для выполнения ЛР0 это оборудование не требуется.

В главах 6–7 STM32F411CEU6 используется как пример целевой платформы для объяснения устройства образа и запуска программы. Готовый образ зависит от набора команд процессора, карты памяти, параметров компоновки и стартового кода; перенос на другую платформу требует адаптации проекта [1, 2].

## 4. Ожидаемый результат

После выполнения лабораторной работы студент должен уметь:

- писать небольшие программы на C, используя переменные, выражения, условия, циклы и функции;

- работать с массивами, указателями, структурами и перечислениями, выбирать способы передачи данных в функции;
- понимать область видимости и время жизни объектов и учитывать их при написании кода;
- читать код, использующий материал главы 1, уверенно объяснять его поведение и находить основные логические ошибки.

Критерий освоения обязательного ядра — уверенная работа с теорией главы 1, понимание материала глав 2–3 и выполненная практика по их темам. При работе с главами 2–3 студент может обращаться к методичке за точными формулировками и деталями применения.

При изучении глав 4–7 студент знакомится с организацией многофайловой программы, сборкой, устройством образа и запуском программы на микроконтроллере. Дополнительный раздел об ООП даёт основу для дальнейшего структурирования данных и организации кода в командных проектах. Эти темы закрепляются по мере их применения в последующих работах.

# Теоретическая часть

## Глава 1. Основы C/C++ и базовые конструкции языка

Практические задания ЛР0 выполняются на сайте SDB Lab; решения необходимо отправлять на сайте. Локальную среду разработки студент использует по своему усмотрению для подготовки, запуска и отладки программ. Её установка не является требованием ЛР0.

Для Windows один из возможных вариантов локальной среды — Visual Studio Code с расширением C/C++ и комплект GCC/GDB из MinGW-w64. Установка и настройка описаны в официальной инструкции Visual Studio Code.

Инструкция использует пример на C++. При локальной сборке примеров этой главы создайте файл `main.c` и используйте компилятор `gcc`, а не `g++`. Примеры написаны на C11. Программу можно собрать в терминале командой:

```
gcc -std=c11 -Wall -Wextra -g -O0 main.c -o main.exe
```

Если вы используете локальную среду, проверьте, что программа собирается и запускается, а отладчик позволяет поставить точку останова и посмотреть значения переменных.

### 1.1. Исходный текст программы: инструкции, блоки, комментарии

Создайте файл `main.c` и запишите в него следующий шаблон:

```c
int main(void)
{
    /* Здесь будем писать код. */

    return 0;
}
```

Код первых примеров размещайте внутри фигурных скобок, перед строкой `return 0;`. Пока используйте этот шаблон без изменений.

**Инструкция** (statement) задаёт действие или управляет порядком выполнения программы. Например, `count = 5;` присваивает переменной `count` значение `5`. Внутри блока инструкции выполняются последовательно, пока программа не встретит конструкцию, изменяющую порядок выполнения.

В теле функции встречаются также **объявления**, которые вводят имена и описывают типы объектов. Объявление переменной и инструкция присваивания имеют разное назначение:

```c
int main(void)
{
    int count = 3;  /* Объявление с начальным значением. */
    count = 5;      /* Изменение значения существующей переменной. */
    return 0;
}
```

После объявления `count` имеет значение `3`; после присваивания — `5`. Точка с запятой завершает это объявление и инструкции присваивания и возврата. Объявление или инструкция могут занимать несколько строк; перенос строки их не завершает.

**Блок**, или составная инструкция, — последовательность объявлений и инструкций между фигурными скобками. Тело `main` является блоком; внутри него можно разместить другие блоки.

```c
int main(void)
{
    int count = 3;
    {
        count = 5;
    }
    return 0;
}
```

## Читаемость и организация кода

Код должен быть понятен человеку, который его читает, проверяет и изменяет, в том числе самому автору спустя некоторое время. Читаемость — одно из основных требований к программе: по тексту должно быть видно, какие данные обрабатываются, в каком порядке выполняются действия и для чего они нужны.

Используйте содержательные имена, единообразные отступы и пустые строки между логически связанными группами действий. В этой методичке принят отступ в четыре пробела внутри каждого блока. Комментариями поясняйте назначение действий и причины решений; очевидную операцию лучше выразить понятным именем.

Пробелы и переносы строк обычно не меняют смысл при сохранении разделения элементов: `int count = 3;` и `int count=3;` эквивалентны, а `intcount=3;` содержит другое имя. Для читателя расположение кода существенно: отступы должны отражать вложенность блоков.

В практических заданиях ЛР0 студент самостоятельно определяет логику и структуру программы, выбирает интерфейсы и пишет код. ИИ может пояснять конструкции языка и помогать разбирать ошибки. Студент должен уметь читать, проверять и объяснять написанный код: успешная компиляция ещё не означает, что программа решает поставленную задачу.

**Комментарии** содержат пояснения для читателя и не задают выполняемых действий. В C11 доступны два способа записи:

```
/* Комментарий между парными ограничителями. */

/* Такой комментарий
   может занимать несколько строк. */

// Комментарий до конца текущей строки.
```

Комментарии вида `/* ... */` нельзя вкладывать друг в друга: первый встретившийся ограничитель `*/` завершает комментарий. Комментарий используют для пояснения назначения кода или причины выбранного решения.

Если в примере показано только несколько строк кода, вставляйте их в указанное место шаблона. Значения переменных смотрите в отладчике, установив точку останова после изучаемых инструкций.

## 1.2. Переменные, инициализация и основные типы данных

**Объект** в терминологии C — область хранения данных, содержимое которой представляет некоторое значение. В первых примерах объект получает имя через объявление переменной. **Тип** определяет допустимые значения и правила операций над ними. Имя позволяет обращаться к объекту в исходном тексте.

```
int main(void)
{
    int temperature = 24;
    temperature = 25;
    return 0;
}
```

В объявлении `int temperature = 24;` тип — `int`, имя — `temperature`, начальное значение — `24`. Установка начального значения при создании объекта называется **инициализацией**. В следующей строке объект уже существует, поэтому `temperature = 25;` является присваиванием. Инициализация и присваивание не являются взаимозаменяемыми понятиями, хотя в этих примерах обе записи содержат знак `=`.

Имя переменной может содержать латинские буквы, цифры и символ подчёркивания, но не должно начинаться с цифры. Для учебных программ рекомендуется именно этот набор символов. Регистр учитывается: `count` и `Count` — разные имена. Ключевые слова, например `int` и `return`, использовать как имена нельзя. Имена должны обозначать смысл данных: `sample_count` обычно полезнее, чем `x`. Для собственных имён следует избегать начального подчёркивания: часть таких имён зарезервирована стандартом.

**Неинициализированная локальная переменная** обычного вида не получает автоматически нулевое значение:

```
/* Фрагмент внутри функции: значение ещё не задано. */
int count;
```

До чтения этой переменной ей необходимо присвоить значение. Чтение её неопределённого значения может приводить к неопределённому поведению. Инициализируйте переменную при объявлении, если начальное значение известно. Правила инициализации других объектов рассматриваются в главе 3.

## Целые типы

Целые типы представляют целочисленные значения в конечном диапазоне. Основной знаковый тип — `int`; он может представлять отрицательные значения, ноль и положительные значения. Запись `unsigned int`, сокращённо `unsigned`, обозначает соответствующий беззнаковый тип, который представляет ноль и положительные значения.

```
int main(void)
{
    int offset = -12;
    unsigned int sample_count = 20u;
    return 0;
}
```

Суффикс `u` в `20u` обозначает беззнаковую целочисленную константу. При выборе типа учитывайте диапазон исходных данных и результатов вычислений. Правила переполнения рассматриваются в 1.3.

Кроме `int`, существуют `short`, `long`, `long long` и их беззнаковые варианты. Их точные размеры зависят от реализации языка: нельзя в общем случае утверждать, что `int` всегда занимает четыре байта. Стандарт задаёт минимальные диапазоны и соотношения между типами. Конкретные ограничения целых типов предоставляются заголовком `<limits.h>`, например через `INT_MIN`, `INT_MAX` и `UINT_MAX`.

**Заголовочный файл** предоставляет объявления типов и других имён, необходимые исходному файлу. Директива `#include <limits.h>` подключает стандартный заголовок; она обрабатывается перед собственно компиляцией и не является выполняемой инструкцией. Имена `INT_MIN` и `INT_MAX` — макросы стандартной библиотеки, представляющие границы диапазона `int`.

```
#include <limits.h>

int main(void)
{
    int minimum = INT_MIN;
    int maximum = INT_MAX;
    unsigned int unsigned_maximum = UINT_MAX;
    return 0;
}
```

## Символьные типы и размер объекта

Тип `char` используется для хранения символов и малых целых значений. Символьная константа записывается в одинарных кавычках:

```
int main(void)
{
    char marker = 'A';
    unsigned char small_value = 200;
    return 0;
}
```

`'A'` в C имеет тип `int`; при инициализации значение преобразуется к `char`. Конкретный числовой код символа зависит от используемого набора символов. Обычный `char`,

`signed char` и `unsigned char` — три разных типа. Будет ли обычный `char` вести себя как знаковый или беззнаковый при представлении значений, зависит от реализации.

Оператор `sizeof` определяет размер типа или объекта в **байтах C**. Один такой байт — размер `char`, поэтому `sizeof(char)` всегда равно `1`. Количество битов в байте задаётся `CHAR_BIT` из `<limits.h>`; стандарт гарантирует не менее восьми битов.

```c
#include <stddef.h>

int main(void)
{
    int count = 3;
    size_t object_size = sizeof count;
    size_t type_size = sizeof(int);
    return 0;
}
```

Результат `sizeof` имеет специальный беззнаковый целочисленный тип `size_t`; его имя доступно, в частности, через `<stddef.h>`. Для типа используются скобки: `sizeof(int)`. Для объекта допустимы `sizeof count` и `sizeof(count)`. Размер в байтах не следует путать с текущим значением переменной.

## Типы с заданной шириной

При работе с протоколами и встроенными системами часто требуется явно указать ширину целого значения. Заголовок `<stdint.h>` предоставляет имена таких типов: например, `uint8_t`, `int16_t`, `uint32_t`. Число обозначает ширину в битах; префикс `u` — отсутствие знака.

```c
#include <stdint.h>

int main(void)
{
    uint8_t level = 200;
    int16_t correction = -120;
    uint32_t counter = 1000;
    return 0;
}
```

Типы с точной шириной предусмотрены только в реализациях, где существуют соответствующие типы без дополнительных битов заполнения. В стандартных средствах разработки для STM32 перечисленные имена доступны. Ширина объекта при этом не обязательно совпадает с шириной промежуточного результата выражения: перед операцией типы могут преобразовываться.

## Типы с плавающей точкой

Типы `float` и `double` предназначены для представления чисел с дробной частью и большого диапазона порядков. Набор представимых значений конечен, поэтому большинство действительных чисел сохраняется приближённо. Тип `double` обеспечивает точность не ниже, чем `float`; точные характеристики определяет реализация. Дополнительный тип `long double` в первых заданиях не требуется.

```c
int main(void)
{
    float voltage = 3.3f;
    double interval = 0.125;
    return 0;
}
```

Десятичный разделитель в исходном коде — точка. Константа `3.3f` имеет тип `float`, а `0.125` без суффикса — тип `double`. Буква `f` относится к типу константы, а не к имени переменной. Наличие дробной записи не означает точного хранения: например, десятичное значение `3.3` при обычном двоичном представлении хранится приближённо. Тип выбирают с учётом требуемой точности и диапазона значений.

## Логический тип

Логические данные обозначают два состояния: ложь и истину. В C11 встроенный тип называется `_Bool`. При преобразовании значения к этому типу ноль становится `0`, а ненулевое значение — `1`.

Заголовок `<stdbool.h>` предоставляет удобные имена `bool`, `false` и `true`. В C11 это макросы: `bool` соответствует `_Bool`, а `false` и `true` — значениям `0` и `1`.

```c
#include <stdbool.h>

int main(void)
{
    bool ready = false;
    ready = true;
    return 0;
}
```

Логические значения используются для хранения признаков и результатов проверок.

Тип следует выбирать по смыслу данных и требуемому диапазону. Количество элементов обычно целочисленное, физическая величина с дробной частью может требовать типа с плавающей точкой, признак готовности — логического типа. Проверьте, что выбранный тип представляет все необходимые значения.

## Сводная таблица типов

Ниже приведены размеры и диапазоны для целевой платформы курса: STM32F411, Arm GNU Toolchain и стандартного ABI AAPCS32. Здесь один байт содержит 8 битов. Это характеристики выбранной платформы, а не универсальные требования C/C++: на компьютере размеры `long`, `long double` и некоторых других типов могут отличаться. Размер проверяют через `sizeof`, границы целых типов — через `<limits.h>` и `<stdint.h>`, характеристики вещественных — через `<float.h>`.

| Тип | Байт | Минимум | Максимум |
|---|---|---|---|
| `char` | 1 | 0 | 255 |
| `signed char` | 1 | −128 | 127 |
| `unsigned char` | 1 | 0 | 255 |
| `short` | 2 | −32768 | 32767 |
| `unsigned short` | 2 | 0 | 65535 |
| `int`, `long` | 4 | −2147483648 | 2147483647 |
| `unsigned int`, `unsigned long` | 4 | 0 | 4294967295 |
| `long long` | 8 | −9223372036854775808 | 9223372036854775807 |
| `unsigned long long` | 8 | 0 | 18446744073709551615 |
| `_Bool` в C, `bool` в C++ | 1 | 0 (ложь) | 1 (истина) |
| `float` | 4 | $\approx -3.4028235 \times 10^{38}$ | $\approx +3.4028235 \times 10^{38}$ |
| `double`, `long double` | 8 | $\approx -1.7976931 \times 10^{308}$ | $\approx +1.7976931 \times 10^{308}$ |
| `int8_t`, `uint8_t` | 1 | −128 / 0 | 127 / 255 |
| `int16_t`, `uint16_t` | 2 | −32768 / 0 | 32767 / 65535 |
| `int32_t`, `uint32_t` | 4 | −2147483648 / 0 | 2147483647 / 4294967295 |
| `int64_t`, `uint64_t` | 8 | −9223372036854775808 / 0 | 9223372036854775807 / 18446744073709551615 |
| `size_t` | 4 | 0 | 4294967295 |

Таблица 1: Сводная таблица типов

Для пар `intN_t`, `uintN_t` первое число относится к знаковому типу, второе — к беззнаковому. Обычный `char` в стандартной конфигурации AAPCS32 беззнаковый; параметр компилятора `-fsigned-char` меняет его диапазон на −128...127. В C11 имя `bool` из `<stdbool.h>` обозначает `_Bool`.

У `float` и `double` в таблице указаны крайние конечные значения, а не точность и не минимальное положительное число. Оба типа представляют также ноль и дробные числа между приведёнными границами, но не каждое действительное число. Для обычных форматов IEEE 754 на этой платформе наименьшие положительные нормализованные значения приблизительно равны $1.1754944 \times 10^{-38}$ и $2.2250739 \times 10^{-308}$; существуют и меньшие положительные субнормальные значения. `float` обеспечивает около 6–7 значащих десятичных цифр, `double` — около 15–16.

Размеры базовых типов определены в AAPCS32, разделы 5.1 и 8.1.1. Диапазоны целых типов соответствуют их разрядности на выбранной платформе.

## 1.3. Выражения, операторы и преобразования типов

**Выражение** — конструкция языка, которая задаёт вычисление значения, обозначает объект или вызывает действие. Простейшие выражения — литерал `12` и имя переменной `count`. Более сложные выражения строятся с помощью операторов: например, `count + 1`.

**Оператор** в контексте выражений — обозначение операции над одним или несколькими операндами. В выражении `a + b` оператором является `+`, а операндами — `a` и `b`. Унарный оператор имеет один операнд, бинарный — два. Например, `-a` содержит унарный минус, а `a - b` — бинарный оператор вычитания.

Тип выражения определяет, какие значения оно может представлять и какие операции с ним допустимы. В частности, одинаковая запись операции может означать разные вычисления для целочисленных типов и типов с плавающей точкой.

Выражение может иметь **побочный эффект** — изменение состояния программы в дополнение к вычислению значения. Присваивание изменяет переменную; сложение само по себе этого не делает.

```
int count = 3;
int next = count + 1;  /* next равно 4; count по-прежнему равно 3 */
count = count + 1;     /* теперь count равно 4 */
```

Директивы `#include` размещайте в начале файла перед `main`, остальные строки фрагментов — внутри `main` перед `return 0;`.

### Арифметические операции

Основные арифметические операторы:

| Оператор | Операция | Пример |
|---|---|---|
| `+` | Сложение | `a + b` |
| `-` | Вычитание | `a - b` |
| `*` | Умножение | `a * b` |
| `/` | Деление | `a / b` |
| `%` | Остаток от целочисленного деления | `a % b` |
| Унарный `-` | Изменение знака | `-a` |
| Унарный `+` | Числовое значение после целочисленного продвижения | `+a` |

Таблица 2: Арифметические операции

Если оба операнда `/` целочисленные, выполняется **целочисленное деление**: дробная часть результата отбрасывается в сторону нуля. Сохранение результата в переменной с плавающей точкой не меняет вид уже выполненного деления.

```
int a = 7;
int b = 2;
int quotient = a / b;         /* 3 */
double first = a / b;         /* 3.0: сначала целочисленное деление */
double second = (double)a / b; /* 3.5: деление с плавающей точкой */
```

В последней строке `(double)a` явно преобразует значение `a` к типу `double`. Поэтому перед делением значение `b` тоже преобразуется к `double`.

Оператор `%` допустим только для целочисленных операндов. Для представимого частного и ненулевого делителя выполняется соотношение:

```
a == (a / b) * b + a % b
```

В C11 ненулевой остаток имеет знак делимого:

```
int q1 =  7 / 3;  /*  2 */
int r1 =  7 % 3;  /*  1 */
int q2 = -7 / 3;  /* -2 */
int r2 = -7 % 3;  /* -1 */
```

Поэтому `%` нельзя без дополнительной обработки считать операцией, всегда возвращающей неотрицательный математический остаток.

Деление целых чисел на ноль и получение остатка при нулевом делителе приводят к неопределённому поведению. В этой работе делитель должен быть ненулевым и для вычислений с плавающей точкой: рассчитывать на бесконечность или другое специальное значение без учёта правил конкретной реализации не следует.

## Сравнения и логические операции

Операторы сравнения проверяют отношение между значениями:

| Оператор | Условие истинности |
|---|---|
| `==` | Значения равны |
| `!=` | Значения различаются |
| `<` | Левое значение меньше правого |
| `<=` | Левое значение меньше или равно правому |
| `>` | Левое значение больше правого |
| `>=` | Левое значение больше или равно правому |

Таблица 3: Сравнения и логические операции

В C результат сравнения имеет тип `int` и равен `1`, если отношение истинно, или `0`, если оно ложно.

```
int temperature = 25;
int above_limit = temperature > 30; /* 0 */
int equal_value = temperature == 25; /* 1 */
```

Оператор == проверяет равенство, а = выполняет присваивание. Подмена одного другим меняет смысл программы.

Логические операторы трактуют нулевое значение как ложь, а ненулевое — как истину. Для рассматриваемых здесь числовых операндов это означает, что и `1`, и `-3`, и `2.5` логически истинны.

| Оператор | Назначение |
|---|---|
| `!a` | Логическое НЕ: истинно, если `a` равно нулю |
| `a && b` | Логическое И: истинно, если оба операнда ненулевые |
| `a \|\| b` | Логическое ИЛИ: истинно, если хотя бы один операнд ненулевой |

Таблица 4: Сравнения и логические операции

Результаты этих операторов также имеют тип `int` и значения `0` или `1`.

```
int temperature = 25;
int within_range = temperature >= 10 && temperature <= 30; /* 1 */
int outside_range = temperature < 10 || temperature > 30;  /* 0 */
int disabled = !within_range;                              /* 0 */
```

Математическая запись `10 <= temperature <= 30` не проверяет принадлежность интервалу. В C она разбирается как `(10 <= temperature) <= 30`: второе сравнение получает слева уже `0` или `1`. Для проверки интервала нужны два сравнения, соединённые `&&`.

Операторы `&&` и `||` используют **короткое замыкание**: правый операнд вычисляется только тогда, когда он нужен для определения результата. У `&&` вычисление прекращается, если левый операнд ложен; у `||` — если левый операнд истинен. Левый операнд вычисляется раньше правого.

```
int divisor = 0;
int valid = divisor != 0 && 12 / divisor > 2;
/* valid равно 0; выражение 12 / divisor не вычисляется */
```

Эта гарантия позволяет поставить проверку допустимости перед потенциально ошибочной операцией. Побитовые операторы `&` и `|` имеют другое назначение и не заменяют `&&` и `||`; они рассматриваются в главе 2.

## Присваивание и изменение значения

Присваивание `a = b` вычисляет правый операнд, преобразует его к типу левого объекта и сохраняет результат в этом объекте. Левый операнд должен обозначать объект, который разрешено изменять: например, обычную переменную. Литералу или результату сложения присвоить значение нельзя.

```
int count = 2;
count = 5;
count = count + 3; /* 8 */
```

Составные операторы присваивания объединяют вычисление и сохранение результата:

```
int count = 10;
count += 3; /* 13 */
count -= 2; /* 11 */
count *= 2; /* 22 */
count /= 4; /* 5: целочисленное деление */
count %= 3; /* 2 */
```

Для простой переменной запись `count += 3` даёт тот же результат, что `count = count + 3`. В общем случае составное присваивание вычисляет левый операнд один раз.

Операторы `++` и `--` увеличивают или уменьшают значение на единицу. У них есть префиксная и постфиксная формы:

```
int count = 4;
int first = ++count;  /* count равно 5; first равно 5 */
int second = count++; /* second равно 5; count равно 6 */
```

Префиксная форма возвращает обновлённое значение, постфиксная — значение до изменения. Если результат выражения не используется, как в отдельной инструкции `count++;`, обе формы дают одинаковое итоговое значение переменной.

Присваивание тоже является выражением и имеет значение, однако для начальных программ рекомендуется записывать изменения переменных отдельными инструкциями. Это делает последовательность действий явной.

## Приоритет, группировка и порядок вычисления

Приоритет операторов определяет, как выражение группируется при отсутствии скобок. Для рассмотренных операторов порядок от более высокого приоритета к более низкому следующий:

1. Постфиксные `++` и `--`.
2. Унарные `+`, `-`, `!`, префиксные `++` и `--`, приведение типа.
3. `*`, `/`, `%`.
4. Бинарные `+`, `-`.
5. `<`, `<=`, `>`, `>=`.
6. `==`, `!=`.
7. `&&`.
8. `||`.
9. `=`, `+=`, `-=`, `*=`, `/=`, `%=`.

Это сокращённый перечень, а не таблица всех операторов C. Скобки позволяют явно задать группировку:

```
int first = 2 + 3 * 4;   /* 14: 2 + (3 * 4) */
int second = (2 + 3) * 4; /* 20 */
int third = 20 / 5 * 2;  /* 8: (20 / 5) * 2 */
```

Арифметические бинарные операторы одного уровня группируются слева направо. Присваивания группируются справа налево: `a = b = 3` означает `a = (b = 3)`. Такая характеристика называется **ассоциативностью** оператора.

Приоритет и ассоциативность описывают структуру выражения, но не задают общий порядок вычисления его операндов. Например, из записи `a + b` не следует универсальная гарантия «сначала вычислить `a`, затем `b`». Скобки также не устанавливают такой порядок.

Изменение переменной и другие обращения к ней внутри одного выражения требуют установленного языком порядка. Например, выражения `count++ + count++` и `count = count++` имеют неопределённое поведение в C11. Вместо подобных конструкций записывайте отдельные инструкции.

## Неявные преобразования типов

В C преобразования типов могут происходить автоматически: при арифметических операциях, сравнениях, присваивании и других действиях. Это **неявные преобразования**. Они позволяют совместно использовать разные типы, но могут изменить значение или способ выполнения операции.

Перед многими операциями значения типов `char`, `signed char`, `unsigned char`, `short` и `unsigned short` проходят **целочисленное продвижение**: преобразуются к `int`, если он представляет все значения исходного типа, иначе к `unsigned int`. Поэтому выражение с маленькими целочисленными операндами не обязательно вычисляется в их исходном типе.

```
unsigned char a = 10;
unsigned char b = 20;
int sum = a + b; /* когда int представляет весь диапазон unsigned char */
```

Для многих бинарных арифметических операций и сравнений затем выбирается общий тип операндов. Это называется **обычными арифметическими преобразованиями**. Для базовых случаев:

- Если один операнд имеет тип `double`, а другой — целочисленный тип или `float`, вычисление выполняется в `double`.
- Если один операнд имеет тип `float`, а другой — целочисленный тип, вычисление выполняется в `float`.
- Для двух целочисленных операндов после продвижения общий тип зависит от ранга (установленного языком порядка типов), знаковости и представимых диапазонов. Он выбирается по правилам языка, а не по текущим значениям переменных.

В частности, `int` и `unsigned int` при совместной операции преобразуются к `unsigned int`. Отрицательное значение при этом становится большим неотрицательным значением:

```
int signed_value = -1;
unsigned int unsigned_value = 1;
int less = signed_value < unsigned_value; /* 0 */
```

Результат может противоречить ожидаемому сравнению математических чисел. Поэтому при смешении знаковых и беззнаковых типов необходимо проверить допустимые диапазоны и понять, в каком типе выполняется операция. Безусловное добавление приведения типа не исправляет ошибку выбора диапазона.

При присваивании значение преобразуется к типу назначения независимо от типа, в котором выполнялось исходное вычисление:

```
double source = 3.9;
int whole = source; /* 3: дробная часть отбрасывается */
double restored = whole; /* 3.0: утраченная дробная часть не восстанавливается */
```

Преобразование конечного числа с плавающей точкой в целое допустимо только тогда, когда результат после отбрасывания дробной части представим в целевом типе. Иначе поведение не определено. Преобразование целого в тип с плавающей точкой тоже может потерять точность, если исходное значение нельзя представить точно.

## Явное приведение типа

**Приведение типа** явно задаёт преобразование значения выражения:

```
(тип) выражение
```

```c
int completed = 3;
int total = 4;
double fraction = (double)completed / total; /* 0.75 */
double lost = (double)(completed / total);   /* 0.0 */
```

В первом случае тип изменён до деления, во втором — после целочисленного деления. Приведение не изменяет тип или значение исходной переменной и не возвращает информацию, утраченную при предыдущем вычислении.

Явная запись преобразования не гарантирует его безопасность. Перед сужением диапазона нужно убедиться, что значение представимо в целевом типе. Преобразование непредставимого значения в знаковый целочисленный тип зависит от реализации; для начальных программ таких преобразований следует избегать.

## Диапазоны, переполнение и точность

У каждого целочисленного типа есть ограниченный диапазон. Для знакового целого типа **переполнение** возникает, когда математический результат не представим в типе вычисления.

Для знаковой целочисленной арифметики переполнение приводит к **неопределённому поведению**: стандарт C не устанавливает требований к поведению программы при такой операции. Нельзя рассчитывать на переход от максимального положительного числа к отрицательному, даже если такой результат наблюдался в конкретном запуске. Другая оптимизация компилятора может изменить проявление ошибки.

```c
#include <limits.h>
int value = INT_MAX;
/* value + 1 вычислять нельзя: переполнение типа int */
```

Неопределённое поведение возможно и без увеличения числа: например, при отрицании минимального значения знакового типа, если положительный результат не представим. Если модуль `INT_MIN` не представим в `int`, деление `INT_MIN / -1` недопустимо; `%` с той же парой операндов также недопустим.

Беззнаковая арифметика определена по модулю числа значений соответствующего типа. Для типа с `N` битами, используемыми для представления значения, результат приводится по модулю `2^N`: например, увеличение максимального значения `unsigned int` на единицу даёт ноль. Преобразование целого числа в беззнаковый тип следует тому же правилу приведения по модулю.

```c
#include <limits.h>
unsigned int value = UINT_MAX;
value += 1u; /* 0: определённое поведение */
```

Такая определённость не означает, что результат соответствует задаче: счётчик, неожиданно вернувшийся к нулю, может быть логической ошибкой. Кроме того, пра-

вило относится к типу фактического вычисления. Целочисленные продвижения могут привести к тому, что операция над небольшими беззнаковыми типами выполняется в знаковом `int`.

Типы с плавающей точкой имеют ограниченную точность. Многие десятичные дроби, например `0.1`, невозможно представить точно в обычном двоичном формате. Поэтому сохранённые значения и результаты операций округляются. Следствием становятся погрешности и зависимость результата от группировки вычислений.

Не следует без анализа задачи проверять точное равенство результатов вычислений с плавающей точкой. Если требуется сравнение с допуском, допустимая погрешность выбирается по смыслу задачи и масштабу значений. Единого малого числа, подходящего для всех таких сравнений, нет. Например, для известного диапазона значений можно проверить попадание в интервал:

```
double measured = 0.1 + 0.2;
double expected = 0.3;
double tolerance = 0.000001; /* выбранный для этого примера абсолютный допуск */
int close = measured >= expected - tolerance
         && measured <= expected + tolerance;
```

Для написания небольших программ важно проверять три вещи: в каком типе выполняется операция, представим ли результат в этом типе и что произойдёт при его сохранении в переменную назначения. Тип переменной слева от `=` сам по себе не расширяет диапазон вычислений справа.

### Пример: обработка одного измерения

Пусть источник данных передаёт температуру в десятых долях градуса: значение 253 означает 25,3 °C. Требуется получить температуру в градусах и определить достижение порога 27 °C.

```
int raw_temperature = 253;
double temperature_c = raw_temperature / 10.0;
double alarm_threshold_c = 27.0;
int alarm = temperature_c >= alarm_threshold_c;
```

Делитель `10.0` обеспечивает вычисление дробной части. Получаем `temperature_c` приблизительно 25.3 и `alarm`, равный 0. Суффикс `_c` в именах обозначает единицу измерения — градусы Цельсия.

## 1.4. Условия и циклы

По умолчанию инструкции выполняются в порядке их записи. **Условные конструкции** позволяют выбирать, какие инструкции выполнить, а **циклы** — повторять выполнение инструкций. Такое изменение порядка выполнения называют управлением потоком выполнения программы.

Фрагменты этой подглавы размещайте внутри `main`. Каждый пример рассматривается независимо; значения переменных наблюдайте в отладчике.

## Условие и конструкция `if`

Условие — выражение, значение которого определяет выбор действия. В C нулевое значение означает ложь, любое ненулевое значение — истину. Поэтому условие не обязано иметь тип `_Bool`: например, целое число `-3` в условии также означает истину. Операторы сравнения и логические операторы дают целочисленный результат `0` или `1`.

Конструкция `if` сначала вычисляет условие. Если оно истинно, выполняется следующая за ним инструкция; если ложно — эта инструкция пропускается. Блок в фигурных скобках является одной составной инструкцией и позволяет объединить несколько действий.

```
int value = -7;
int magnitude = value;

if (value < 0) {
    magnitude = -value;
}
```

В этом фрагменте выражение `value < 0` истинно, поэтому `magnitude` становится равной `7`. При исходном `value = 5` блок был бы пропущен, и `magnitude` осталась бы равной `5`. Здесь используются небольшие значения: такой способ вычисления модуля нельзя без дополнительной проверки применять ко всему диапазону `int`, поскольку модуль минимального отрицательного значения может не представляться этим типом.

Если нужно выбрать одно из двух действий, используют `else`:

```
int temperature = 24;
int high_temperature;

if (temperature > 30) {
    high_temperature = 1;
} else {
    high_temperature = 0;
}
```

Выполняется ровно одна из двух ветвей. При `temperature = 24` результат равен `0`. После выбранной ветви программа продолжает выполнение с инструкции, расположенной после всей конструкции.

Для нескольких последовательных проверок удобно записывать цепочку:

```
int value = 0;
int sign;

if (value < 0) {
    sign = -1;
} else if (value > 0) {
    sign = 1;
} else {
    sign = 0;
}
```

Условия проверяются сверху вниз, пока не найдена подходящая ветвь. После выполнения этой ветви последующие проверки в цепочке не выполняются. Запись `else if` — это `if`, помещённый в ветвь `else`, а не отдельное ключевое слово языка.

Два независимых `if` имеют другой смысл: второе условие проверяется независимо от результата первого. В частности, первая проверка может изменить переменную, которую затем использует вторая проверка.

## Составные условия и вложенность

Логические операторы объединяют условия: `&&` требует истинности обоих операндов, `||` — хотя бы одного, а `!` меняет истинность на противоположную.

```
int value = 8;
int inside = 0;

if (value >= 0 && value < 10) {
    inside = 1;
}
```

Здесь проверяется полуинтервал от `0` включительно до `10` исключительно. Для проверки обеих границ используются два сравнения, соединённые `&&`.

Короткое замыкание позволяет проверить допустимость операции до её выполнения:

```
int numerator = 12;
int denominator = 0;
int positive_quotient = 0;

if (denominator != 0 && numerator / denominator > 0) {
    positive_quotient = 1;
}
```

При нулевом `denominator` деление не выполняется. Для данного примера ненулевого знаменателя достаточно; в общем случае дополнительно учитывают представимость результата целочисленного деления.

Условные конструкции можно вкладывать друг в друга:

```
int enabled = 1;
int value = 12;
int accepted = 0;

if (enabled) {
    if (value >= 0 && value <= 100) {
        accepted = 1;
    }
}
```

Внутреннее условие проверяется только при истинном внешнем. Если фигурные скобки опущены, `else` относится к ближайшему предшествующему `if`, для которого ещё нет `else`. Поэтому в учебном и рабочем коде рекомендуется явно обозначать блоки фигурными скобками, даже если в ветви пока одна инструкция. Отступы помогают читать текст, но не определяют его смысл для компилятора.

## Выбор по значению: `switch`

Конструкция `switch` удобна, когда нужно выбрать действие по одному целочисленному значению. Управляющее выражение должно иметь целочисленный тип. Для про-

верки произвольных диапазонов и вещественных значений используют `if`.

```
int mode = 2;
int limit = 0;

switch (mode) {
    case 0:
        limit = 10;
        break;
    case 1:
        limit = 20;
        break;
    case 2:
        limit = 30;
        break;
    default:
        limit = 5;
        break;
}
```

Значение `mode` вычисляется один раз. Выполнение начинается с метки `case`, значение которой совпало с результатом выражения. Здесь выбирается `case 2`, переменная `limit` получает значение `30`, а `break` завершает `switch`.

Значения после `case` должны быть целочисленными константными выражениями; обычная переменная для этого не подходит. Повторяющиеся значения `case` недопустимы. Метка `default` задаёт место начала выполнения при отсутствии совпадения. Она необязательна: если совпадения нет и `default` отсутствует, тело `switch` не выполняется.

Метка `case` сама по себе не завершает предыдущую ветвь. Если после действий нет `break`, выполнение продолжается через следующую метку независимо от её значения. Это называется **сквозным выполнением** (fallthrough):

```
int code = 1;
int result = 0;

switch (code) {
    case 1:
        result += 10;
        /* Намеренно продолжаем выполнение в case 2. */
    case 2:
        result += 20;
        break;
    default:
        result = -1;
        break;
}
```

При `code = 1` результат равен `30`; при `code = 2` — `20`. Пропущенный случайно `break` меняет поведение программы. Намеренный переход следует явно пояснять комментарием. В C11 нет стандартного атрибута `[[fallthrough]]`.

Для общего действия нескольких значений можно поставить метки подряд:

```
int mode = 1;
int enabled = 0;

switch (mode) {
    case 1:
    case 2:
        enabled = 1;
        break;
    default:
        enabled = 0;
        break;
}
```

## Цикл с предусловием: `while`

**Итерация** — одно выполнение тела цикла. В цикле `while` условие проверяется перед каждой итерацией. Если оно ложно уже при первой проверке, тело не выполнится ни разу.

```
int counter = 0;
int sum = 0;

while (counter < 4) {
    sum += counter;
    counter++;
}
```

Порядок выполнения:

1. Проверяется `counter < 4`.
2. При истинном условии к `sum` прибавляется текущее значение `counter`.
3. `counter` увеличивается на единицу.
4. Выполнение возвращается к проверке условия.
5. При ложном условии цикл завершается.

Тело выполняется для `counter`, равного `0`, `1`, `2` и `3`. После цикла `counter` равна `4`, а `sum` равна `6`. Переменная, используемая в условии, не меняется автоматически: изменение `counter` обеспечивает написанная нами инструкция `counter++`.

Чтобы цикл с изменяемым условием завершался, действия программы должны в какой-то момент сделать условие ложным. Если удалить увеличение счётчика в этом примере, условие останется истинным. Однако оставлять такой ошибочный цикл выполняться не следует: повторное накопление в `sum` в конце концов приведёт к переполнению знакового `int`, а это неопределённое поведение.

## Цикл с постусловием: `do ... while`

Цикл `do ... while` сначала выполняет тело, затем проверяет условие. Поэтому тело выполняется хотя бы один раз.

```
int value = 20;
int iterations = 0;

do {
    value /= 2;
    iterations++;
} while (value > 5);
```

После первой итерации `value` равна `10`, условие истинно. После второй `value` равна `5`, условие ложно. Итог: `value = 5`, `iterations = 2`.

При исходном `value = 4` тело всё равно выполнится один раз и получится `value = 2`. Аналогичный цикл `while (value > 5)` при таком исходном значении не выполнил бы тело ни разу. Точка с запятой после завершающего `while (...)` в конструкции `do ... while` обязательна.

## Цикл `for` и границы повторения

Цикл `for` объединяет начальное действие, условие продолжения и действие после итерации:

```
int sum = 0;

for (int i = 0; i < 5; i++) {
    sum += i;
}
```

Порядок выполнения:

1. `int i = 0` выполняется один раз перед первой проверкой условия.
2. Проверяется `i < 5`.
3. При истинном условии выполняется тело.
4. Выполняется `i++`.
5. Выполнение возвращается к проверке условия.

Тело выполняется пять раз, для `i` от `0` до `4`; итоговая `sum` равна `10`. Объявленная в заголовке переменная `i` доступна в условии, действии после итерации и теле этого цикла, но недоступна после всей конструкции. Такая запись допустима в C11.

Если значение счётчика нужно после цикла, его объявляют заранее:

```
int i;
int sum = 0;

for (i = 0; i < 5; i++) {
    sum += i;
}
/* Здесь i равна 5, sum равна 10. */
```

При записи границ нужно определить, включается ли конечное значение. Для счётчика, начинающегося с `0`, условие `i < 5` задаёт пять итераций, а `i <= 5` — шесть. Ошибка в выборе `<` или `<=` называется ошибкой на единицу.

Тип счётчика должен представлять начальное значение, все используемые промежуточные значения и значение после последнего изменения. Если беззнаковый счётчик,

равный нулю, уменьшить на единицу, он получит максимальное значение своего типа. Например, условие `i >= 0` для беззнакового `i` всегда истинно и не подходит для завершения обратного отсчёта.

Вложенные циклы позволяют повторять группу повторений. При каждом новом входе во внутренний цикл его начальное действие выполняется заново:

```
int iterations = 0;

for (int row = 0; row < 3; row++) {
    for (int column = 0; column < 2; column++) {
        iterations++;
    }
}
```

Внешний цикл выполняет три итерации. На каждой из них внутренний выполняет две. Итоговая `iterations` равна `6`.

## Досрочное завершение и переход к следующей итерации

`break` завершает ближайший охватывающий цикл или `switch`. Внутри вложенного цикла он завершает только внутренний цикл.

```
int first = -1;

for (int i = 0; i < 10; i++) {
    if (i * i >= 20) {
        first = i;
        break;
    }
}
```

Первое подходящее значение `i` равно `5`, поэтому `first` получает `5` и цикл сразу завершается. После `break` действие `i++` этой итерации уже не выполняется.

`continue` пропускает оставшуюся часть тела ближайшего охватывающего цикла. В цикле `for` далее выполняется действие после итерации, затем проверяется условие. В `while` выполнение переходит непосредственно к проверке условия; в `do ... while` — к заключительной проверке условия.

```
int sum = 0;

for (int i = 0; i < 6; i++) {
    if (i % 2 == 0) {
        continue;
    }
    sum += i;
}
```

Чётные значения пропускаются; суммируются `1`, `3` и `5`. Итоговая `sum` равна `9`. Увеличение `i` выполняется и после `continue`.

В цикле `while` нужно следить, чтобы `continue` не пропустил изменение переменной, необходимое для завершения. Если изменение находится ниже `continue`, программа может постоянно возвращаться к одному и тому же состоянию.

Если `switch` находится внутри цикла, `break` внутри этого `switch` завершает только `switch`; цикл продолжается. `continue` внутри такого `switch` относится к охватывающему циклу, поскольку сам `switch` циклом не является.

## Бесконечный цикл и программы для микроконтроллеров

Цикл может намеренно не иметь обычного условия завершения:

```
while (1) {
    /* Повторяемая работа программы. */
}
```

Эквивалентная по условию продолжения запись:

```
for (;;) {
    /* Повторяемая работа программы. */
}
```

В `for` можно опустить любую из трёх частей заголовка. Отсутствующее условие продолжения трактуется как истинное. Такой цикл можно завершить инструкцией `break`, если она предусмотрена в его теле; в приведённых фрагментах её нет.

Основной цикл программы микроконтроллера часто выполняется постоянно, повторяя обработку данных и управление устройством.

Бесконечное повторение не делает арифметику безопасной: нельзя без ограничения увеличивать знаковый счётчик и ожидать определённого поведения после переполнения. Также пустой цикл не следует использовать как точно заданную временную задержку: его длительность нельзя определить только по исходному тексту.

## Характерные ошибки

**Присваивание вместо сравнения.** Запись `if (value = 5)` допустима в C: сначала `value` получает `5`, затем это ненулевое значение рассматривается как истинное условие. Для сравнения нужно `if (value == 5)`. Предупреждения компилятора часто помогают заметить ошибку, но допустимость кода ещё не означает правильность его смысла.

**Лишняя точка с запятой после условия.** Следующий фрагмент содержит ошибку логики:

```
int value = -1;
int accepted = 0;

if (value > 0); /* Пустая инструкция является телом if. */
{
    accepted = 1;
}
```

Блок после `if` выполняется безусловно, поэтому `accepted` получает `1` даже при отрицательном `value`.

Аналогично в `while (condition);` и `for (...);` телом цикла становится пустая инструкция. Следующий блок уже не относится к циклу. Пустое тело иногда бывает

намеренным, но на начальном этапе такую запись следует проверять особенно внимательно. Исключение по форме — обязательная точка с запятой в конце `do ... while (condition);`.

**Неверная граница или отсутствие изменения.** Для любого цикла необходимо явно проследить начальное состояние, первую и последнюю выполняемые итерации, изменение состояния и первую ложную проверку. Такая проверка по шагам помогает обнаружить лишнюю итерацию, пропущенное значение и цикл, который не достигает завершения.

## Пример: обработка серии измерений

Продолжим задачу: обработаем пять значений, посчитаем срабатывания по порогу и найдём среднюю температуру. Допустимые входные значения — от 0 до 500 включительно. Значения задаются в коде; на четвёртой итерации подставляется -1 для проверки отбрасывания ошибочных данных.

```
int sample_count = 5;
int valid_count = 0;
int alarm_count = 0;
double temperature_sum_c = 0.0;
double average_temperature_c = 0.0;
double alarm_threshold_c = 27.0;

for (int sample = 0; sample < sample_count; ++sample) {
    int raw_temperature = 240 + 10 * sample;
    if (sample == 3) {
        raw_temperature = -1;
    }

    if (raw_temperature < 0 || raw_temperature > 500) {
        continue;
    }

    double temperature_c = raw_temperature / 10.0;
    temperature_sum_c += temperature_c;
    ++valid_count;

    if (temperature_c >= alarm_threshold_c) {
        ++alarm_count;
    }
}

if (valid_count > 0) {
    average_temperature_c = temperature_sum_c / valid_count;
}
```

В расчёт входят 24, 25, 26 и 28 °C: `valid_count` равен 4, `alarm_count` — 1, средняя температура — 25,75 °C. Проверка `valid_count` предотвращает деление на ноль, если все измерения отброшены. При отсутствии данных значение 0.0 не является измеренной средней температурой: его нужно рассматривать вместе с `valid_count`.

## 1.5. Функции: объявление, определение, параметры и возвращаемое значение

Функция — именованная часть программы, которая выполняет определённую задачу. Функция может получать входные значения через параметры и передавать результат вызывающему коду через возвращаемое значение. Выполнение тела функции начинается при её вызове.

Функции позволяют разделять программу на части: например, отдельно вычислять значение, проверять его допустимость и выбирать дальнейшее действие. Если операция требуется несколько раз, её реализацию можно записать один раз и вызывать в разных местах.

### Объявление и определение функции

Объявление сообщает компилятору имя функции, тип возвращаемого значения и сведения о параметрах. Объявление со списком типов параметров называется прототипом функции:

```
int maximum(int left, int right);
```

Здесь `maximum` — имя функции; первый `int` — тип возвращаемого значения; `left` и `right` — имена двух параметров типа `int`. Точка с запятой завершает объявление. Тела функции здесь нет.

Имена параметров в прототипе можно опустить:

```
int maximum(int, int);
```

Однако имена полезны для чтения: они помогают обозначить назначение входных данных. Имена параметров в объявлении и определении могут различаться, но типы должны быть согласованы.

Определение функции содержит её тело и тем самым задаёт выполняемые действия:

```
int maximum(int left, int right)
{
    if (left > right) {
        return left;
    }

    return right;
}
```

Определение также является объявлением. После заголовка определения записывается блок в фигурных скобках; точка с запятой вместо тела превратила бы эту запись в прототип.

До вызова функции компилятор должен встретить её подходящее объявление. Поэтому можно разместить определение выше вызывающего кода либо записать прототип выше, а определение — ниже:

```c
int maximum(int left, int right);

int main(void)
{
    int a = 12;
    int b = 7;
    int result = maximum(a, b);

    if (result == 12) {
        return 0;
    }

    return 1;
}

int maximum(int left, int right)
{
    if (left > right) {
        return left;
    }

    return right;
}
```

В этом примере `main` использует сведения из прототипа, а тело `maximum` находится ниже. Позднее объявления будут вынесены в заголовочные файлы, чтобы функции можно было использовать из других исходных файлов.

## Параметры и аргументы

Параметры записываются в объявлении и определении функции. Аргументы — выражения, которые передаются при конкретном вызове. В выражении `maximum(a, b)` аргументами являются `a` и `b`, а внутри определения соответствующие параметры называются `left` и `right`.

Вызов записывается как имя функции и заключённый в круглые скобки список аргументов. Аргументы разделяются запятыми. Их количество и типы должны соответствовать объявлению; допустимые неявные преобразования выполняются по правилам языка.

Аргументом может быть переменная, константа или другое выражение:

```c
int first = maximum(3, 8);
int second = maximum(first + 2, 5);
```

Перед началом выполнения тела вычисляются значения аргументов. Для функции с возвращаемым значением результат вызова можно использовать как часть выражения. В примере `first` получает значение `8`, а `second` — значение `10`.

В C порядок вычисления разных аргументов не задаётся. Поэтому не следует писать вызовы, результат которых зависит от предполагаемого порядка изменения одной переменной внутри аргументов. Для учебных примеров аргументы лучше вычислять отдельными инструкциями, если их вычисление меняет состояние программы.

## Передача по значению

В C аргументы передаются по значению. При вызове каждый параметр получает значение соответствующего аргумента и становится отдельным объектом. Изменение параметра само по себе не изменяет переменную вызывающего кода.

```
int increased(int value);

int main(void)
{
    int number = 4;
    int result = increased(number);

    /* number равно 4, result равно 5. */
    if (number == 4 && result == 5) {
        return 0;
    }

    return 1;
}

int increased(int value)
{
    value = value + 1;
    return value;
}
```

Параметр `value` получает значение `4`. Присваивание изменяет именно `value`. Переменная `number` остаётся равной `4`; вычисленное значение `5` передаётся обратно через `return` и используется для инициализации `result`.

Если требуется обновить исходную переменную таким способом, результат следует явно присвоить ей:

```
number = increased(number);
```

Другие способы передачи доступа к данным рассматриваются в главе об указателях.

## Возвращаемое значение и return

Инструкция `return` завершает текущий вызов функции и передаёт управление вызывающему коду. В функции с возвращаемым значением после `return` записывается выражение:

```
return left;
```

Его значение преобразуется к типу возвращаемого значения функции. Например, функция, объявленная с возвращаемым типом `int`, возвращает значение типа `int`. Желательно выбирать выражения согласованных типов, чтобы преобразования не приводили к неожиданной потере данных.

После выполнения `return` последующие инструкции тела в рамках этого вызова не выполняются. Поэтому ранний возврат удобен для обработки отдельных случаев:

```
int absolute_small(int value)
{
    if (value < 0) {
        return -value;
    }

    return value;
}
```

Допустимый аргумент этой функции — значение `int`, модуль которого представим в `int`. Это ограничение входит в **контракт функции**: условия, при которых функция обязуется выдать определённый результат.

Для обычной функции с возвращаемым типом, отличным от `void`, следует обеспечить возврат значения на каждом возможном пути выполнения. Достижение конца такой функции без `return` не даёт определённого результата; использование вызывающим кодом её возвращаемого значения приводит к неопределённому поведению. Для `main` в обычной среде выполнения есть отдельное правило, описанное ниже.

## Функции без параметров и без возвращаемого значения

В используемом здесь C11 отсутствие параметров явно обозначается словом `void` внутри круглых скобок:

```
int initial_value(void)
{
    return 10;
}
```

Вызов такой функции записывается с пустыми скобками:

```
int value = initial_value();
```

Запись объявления `int initial_value();` в C11 не означает прототип функции без параметров: она не сообщает типы и количество параметров. Поэтому для функции без параметров используем именно `(void)`.

Тип `void` перед именем означает, что функция не возвращает значения. Для выхода из неё используется `return;` без выражения. Если выполнение доходит до закрывающей скобки её тела, вызов также заканчивается.

```
void check_temperature(int temperature)
{
    if (temperature < 0) {
        return;
    }

    /* Здесь могут выполняться действия для неотрицательной температуры. */
}
```

Пример иллюстрирует только форму функции и ранний выход; полезные действия здесь ещё не реализованы. Обычно вызов функции с возвращаемым типом `void` записывают отдельной инструкцией:

```
check_temperature(20);
```

Нельзя инициализировать переменную результатом такого вызова: возвращаемого значения у него нет. Функции без результата нужны, когда их назначение — выполнить действие, например изменить доступные им данные или передать команду устройству. Способы реализации таких действий будут вводиться далее.

## Локальные переменные и отдельные вызовы

Параметры и переменные, объявленные внутри тела функции, доступны в соответствующих областях видимости этой функции. Код другой функции не может обратиться к ним по имени. Совпадение имён локальных переменных в разных функциях не означает, что они обозначают один объект.

Каждый вызов создаёт свои параметры и обычные локальные переменные. Такие объекты существуют на протяжении соответствующего выполнения функции или блока; подробные правила времени жизни будут рассмотрены отдельно. Поэтому предыдущий вызов `increased` не сохраняет изменённый параметр для следующего вызова.

Функция может вызывать другие функции. После завершения вложенного вызова выполнение продолжается в вызывающей функции. Вызов функции самой себя называется рекурсией. Он также создаёт отдельные параметры и локальные переменные для нового вызова. Рекурсия требует условия прекращения вызовов и расходует ресурсы на хранение состояния. В практической части рекурсивные вызовы используются в задании 2.1.12 «Быстрая сортировка»; разбор алгоритма приведён в материале к заданию.

## Роль main

В обычной среде выполнения C, например в программе для персонального компьютера, стандартная форма точки входа без аргументов — `int main(void)`. Возврат из первоначального вызова `main` завершает программу и передаёт среде выполнения код завершения. Значение `0` обозначает успешное завершение. Достижение конца тела `main` в такой среде эквивалентно `return 0;`.

В проекте микроконтроллера `main` обычно вызывается стартовым кодом и содержит постоянный цикл обработки. Возврат из `main` передаёт управление стартовому коду; дальнейшее поведение задаёт конкретный проект. Запуск программы на микроконтроллере рассматривается в главе 7.

## Разбиение задачи на функции

Удобная функция имеет понятное назначение, явно определённые входные данные и ожидаемый результат. Например, задача обработки измерения может состоять из проверки диапазона, вычисления скорректированного значения и выбора состояния. Эти операции можно оформить отдельными функциями.

При разбиении программы следует определить:

- какую одну связную задачу выполняет функция;
- какие значения ей нужны и какие ограничения наложены на них;
- какое значение она возвращает либо какое действие выполняет;
- что происходит в граничных случаях.

Имя должно помогать понять назначение вызова. Функция `maximum(a, b)` читается легче, чем повторяющиеся блоки сравнения в нескольких местах. Разделяйте код на функции по смыслу выполняемых задач.

## Пример: выделение обработки измерений в функции

Перенесём проверку входных данных, перевод в градусы и сравнение с порогом в отдельные функции. Температура, равная порогу или выше него, считается перегревом. `main` отвечает за обход серии и накопление результатов.

```
int is_valid_temperature(int raw_temperature);
double to_celsius(int raw_temperature);
int is_overheated(double temperature_c, double threshold_c);

int main(void)
{
    int valid_count = 0;
    int alarm_count = 0;
    double temperature_sum_c = 0.0;
    double average_temperature_c = 0.0;
    double alarm_threshold_c = 27.0;

    for (int sample = 0; sample < 5; ++sample) {
        int raw_temperature = 240 + 10 * sample;
        if (sample == 3) {
            raw_temperature = -1;
        }

        if (!is_valid_temperature(raw_temperature)) {
            continue;
        }

        double temperature_c = to_celsius(raw_temperature);
        temperature_sum_c += temperature_c;
        ++valid_count;

        if (is_overheated(temperature_c, alarm_threshold_c)) {
            ++alarm_count;
        }
    }

    if (valid_count > 0) {
        average_temperature_c = temperature_sum_c / valid_count;
    }

    return 0;
}

int is_valid_temperature(int raw_temperature)
{
    return raw_temperature >= 0 && raw_temperature <= 500;
}

double to_celsius(int raw_temperature)
{
    return raw_temperature / 10.0;
}
```

```c
int is_overheated(double temperature_c, double threshold_c)
{
    return temperature_c >= threshold_c;
}
```

Результаты остались прежними: четыре допустимых измерения, одно срабатывание по порогу, средняя температура 25,75 °C. Функции получают данные через параметры, а результат передают через `return`. По именам вызовов видно последовательность обработки; например, порог можно изменить в `main` без изменения функции сравнения.

## 1.6. Ввод и вывод в консоль

Программа получает исходные данные через **стандартный ввод** и передаёт ответы через **стандартный вывод**. При запуске в терминале ввод обычно поступает с клавиатуры, а вывод отображается в окне терминала. На сайте проверки входные данные передаются программе автоматически, а её вывод сравнивается с ожидаемым ответом.

Для работы со стандартным вводом и выводом в C подключите заголовок `<stdio.h>`. Он содержит объявления функций `printf`, `scanf` и других функций ввода и вывода.

### Вывод в C: printf

Функция `printf` выводит текст и значения по заданному формату:

```c
#include <stdio.h>

int main(void)
{
    int valid_count = 4;
    double average_c = 25.75;

    printf("Hello, World!\n");
    printf("Valid: %d\nAverage: %.2f\n", valid_count, average_c);
    return 0;
}
```

Результат:

```
Hello, World!
Valid: 4
Average: 25.75
```

Первый аргумент — **строка формата**, заключённая в двойные кавычки. Обычный текст выводится как есть. Обозначения с `%` задают места для значений последующих аргументов: `%d` соответствует `valid_count`, а `%.2f` — `average_c`. Аргументы подставляются в порядке записи. Последовательность `\n` задаёт перевод строки; автоматически `printf` его не добавляет. Для вывода знака процента в строке формата используется `%%`. GNU C Library: основы форматированного вывода.

Основные форматы для задач этой работы:

| Тип значения | Формат printf | Формат scanf |
|---|---|---|
| `int` | `%d` | `%d` |
| `unsigned int` | `%u` | `%u` |
| `long` | `%ld` | `%ld` |
| `long long` | `%lld` | `%lld` |
| `float` | `%f` | `%f` |
| `double` | `%f` | `%lf` |
| `size_t` | `%zu` | `%zu` |
| `char`, вывод или чтение символа | `%c` | `%c` |

Таблица 5: Вывод в C: printf

Для `printf` значение типа `float` при передаче аргумента преобразуется к `double`, поэтому оба типа выводятся через `%f`. При вводе `scanf` должна записать результат в объект конкретного типа: для `float` нужен `%f`, а для `double` — `%lf`. Несоответствие формата типу аргумента может приводить к неопределённому поведению. Правила форматов: C11, N1570, разделы 7.21.6.1–7.21.6.4.

В формате `%f` по умолчанию выводятся шесть знаков после десятичной точки. Запись `%.2f` задаёт два знака, `%.6f` — шесть. Форматирование округляет отображаемое значение, но не изменяет переменную и не повышает точность вычислений.

Для вывода двух чисел через пробел используйте `printf("%d %d\n", a, b);`, для вывода на отдельных строках — `printf("%d\n%d\n", a, b);`. Пробелы и переводы строк задаются самой строкой формата.

## Ввод в C: scanf

Следующая программа читает два целых числа и выводит их сумму:

```c
#include <stdio.h>

int main(void)
{
    int a = 0;
    int b = 0;

    if (scanf("%d %d", &a, &b) != 2) {
        return 1;
    }

    printf("%d\n", a + b);
    return 0;
}
```

Для этого примера оба числа должны быть в диапазоне от $-1000$ до 1000. При вводе `5 2` программа выводит `7`.

`scanf` получает **адреса переменных**, в которые необходимо записать прочитанные значения. В выражении `&a` оператор `&` получает адрес объекта `a`. Здесь достаточно пользоваться правилом: для ввода числа или символа в обычную переменную перед её именем записывается `&`. Адреса и указатели подробно рассматриваются в главе 2. `printf`, напротив, получает сами значения: `a`, а не `&a`.

Возвращаемое значение `scanf` — количество успешно выполненных присваиваний. Для двух чисел ожидается `2`, для одного — `1`. Если введено `5 abc`, первый ввод успешен, второй — нет, и результат вызова равен `1`. При завершении ввода до первого преобразования функция может вернуть `EOF`. Проверка `!= 2` охватывает все случаи, когда два значения не получены. GNU C Library: основы форматированного ввода.

Числовые форматы пропускают начальные пробелы, табуляции и переводы строк. Поэтому `scanf("%d %d &a, &b)` прочитает числа как из строки `5 2`, так и из двух отдельных строк. После ввода в терминале нажмите Enter. Если программа ожидает два числа, а введено только одно, она продолжит ждать второе.

Для обычного ввода чисел не добавляйте `\n` в конец строки формата `scanf`: пробельный символ в формате требует пропустить всю следующую последовательность пробельных символов, и при ручном вводе программа может ждать следующий непробельный символ. Перевод строки после числа и так будет пропущен при следующем числовом чтении.

Исходные числа должны помещаться в выбранный тип. Успешный вызов `scanf` также не означает, что выполнены смысловые ограничения задачи: например, допустимость температуры проверяется отдельно.

## Вещественные числа и длинные целые

Пример для задачи о верёвке:

```c
#include <stdio.h>

int main(void)
{
    double length = 0.0;
    if (scanf("%lf", &length) != 1) {
        return 1;
    }

    double remaining_length = length / 4.0;
    printf("%.6f\n", remaining_length);
    return 0;
}
```

Вход `99.50` даёт выход `24.875000`. В задачах вещественные числа вводятся с десятичной точкой. При обычном запуске программы без изменения локали стандартные функции используют именно точку.

Для больших целочисленных результатов выбирайте подходящий тип **до вычисления**, а затем используйте соответствующий формат:

```c
#include <stdio.h>

int main(void)
{
    long long a = 0;
    long long b = 0;
    if (scanf("%lld %lld", &a, &b) != 2) {
        return 1;
    }

    printf("%lld\n%lld\n%lld\n", a - b, a * b, a + b);
    return 0;
}
```

Это решение задачи 1.1.1: входные числа ограничены диапазоном 32-битного знакового `int`, но прочитаны в `long long`, чтобы все три операции выполнялись в более широком типе. Для входа `5 2` выводятся строки `3`, `10` и `7`. Присваивание уже вычисленного произведения двух `int` переменной `long long` не предотвращает переполнение самого произведения.

## Символы и логические значения

Для чтения одного непробельного символа:

```c
char command = '\0';
if (scanf(" %c", &command) != 1) {
    return 1;
}
printf("%c\n", command);
```

Сам формат `%c` не пропускает пробельные символы. Начальный пробел в `" %c"` позволяет пропустить оставшийся перевод строки после ввода числа. Если необходимо прочитать именно следующий символ, включая пробел или перевод строки, используйте `"%c"`.

Для простого вывода готовой строки можно использовать `puts("YES");`: функция сама добавляет перевод строки. В `printf("YES\n");` перевод строки записывается явно. Один символ выводится также через `putchar('@');`, а перевод строки — через `putchar('\n');`.

Для чтения логического значения, заданного как 0 или 1, сначала прочитайте `int`, проверьте допустимость и затем преобразуйте его к `bool`:

```c
int input = 0;
if (scanf("%d", &input) != 1 || (input != 0 && input != 1)) {
    return 1;
}
bool enabled = input != 0;
printf("%d\n", enabled);
```

Для этого фрагмента, кроме `<stdio.h>`, нужен `<stdbool.h>`. В `scanf` нельзя передавать адрес `bool` для формата `%d`: формат требует адрес `int`. При выводе через `printf` значение `bool` преобразуется к `int`, поэтому `%d` выводит 0 или 1. Если задача требует слова, используйте условие и выводите точно заданные строки, например `YES` и `NO`.

## Последовательность значений и отдельная функция обработки

Количество чисел может задаваться первым числом входа. Их можно читать и обрабатывать по одному, не сохраняя всю последовательность. В следующем примере читаются `count` температур в десятых долях градуса. Допустимый диапазон — от 0 до 500 включительно. Требуется вывести количество допустимых показаний и их среднее в градусах. Если допустимых показаний нет, выводится `NO DATA`.

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

bool is_valid_temperature(int raw_temperature)
{
    return raw_temperature >= 0 && raw_temperature <= 500;
}

int main(void)
{
    int count = 0;
    if (scanf("%d", &count) != 1 || count < 1 || count > 1000) {
        return 1;
    }

    int valid_count = 0;
    double sum_c = 0.0;

    for (int i = 0; i < count; ++i) {
        int raw_temperature = 0;
        if (scanf("%d", &raw_temperature) != 1) {
            return 1;
        }

        if (is_valid_temperature(raw_temperature)) {
            sum_c += raw_temperature / 10.0;
            ++valid_count;
        }
    }

    if (valid_count == 0) {
        puts("NO DATA");
    } else {
        printf("%d %.6f\n", valid_count, sum_c / valid_count);
    }
    return 0;
}
```

Вход:

```
5
240 250 260 -1 280
```

Выход:

```
4 25.750000
```

`main` организует ввод, накопление и вывод. Функция `is_valid_temperature` проверяет переданное значение и возвращает ответ; способ получения данных ей не нужен. При

вводе до специального значения вместо счётчика используется условие: после успешного чтения проверяется признак окончания, например 0, и только затем число обрабатывается. Если признак окончания не входит в последовательность, включать его в сумму или максимум нельзя.

## Ввод и вывод в C++

В C++ можно использовать стандартные потоки `std::cin` и `std::cout`, объявленные в `<iostream>`. Оператор `>>` извлекает значение из входного потока, а `<<` вставляет значение в выходной. В этом контексте они выполняют операции ввода и вывода. Форматы `%d` и `%lf` не нужны: операция выбирается по типу переменной. Адрес переменной через `&` также не передаётся. Префикс `std::` указывает на имя в пространстве имён стандартной библиотеки. C++17: стандартные объекты потоков.

```
#include <iomanip>
#include <iostream>

int main()
{
    double length = 0.0;
    if (!(std::cin >> length)) {
        return 1;
    }

    std::cout << std::fixed << std::setprecision(6)
              << length / 4.0 << '\n';
    return 0;
}
```

Это вариант задачи о верёвке на C++. Проверка `!(std::cin >> length)` обнаруживает неуспешное чтение. Несколько чисел читаются цепочкой `std::cin >> a >> b`; обычное числовое чтение пропускает пробельные символы.

`std::fixed` задаёт вывод вещественных чисел без экспоненциальной записи, а `std::setprecision(6)` при таком режиме задаёт шесть знаков после точки. Для `std::setprecision` нужен `<iomanip>`. Без `std::fixed` эта настройка задаёт количество значащих цифр. Настройки сохраняются для последующих операций с потоком, пока не будут изменены. C++17: setprecision.

Для вывода двух целых чисел через пробел используйте `std::cout << a << ' ' << b << '\n';`. Поток не вставляет разделители сам. Значение `bool` по умолчанию выводится как 0 или 1; `std::boolalpha` переключает вывод на `false` и `true`. Строки `YES` и `NO` по-прежнему нужно выводить явно. Для завершения строки достаточно `'\n'`; `std::endl` дополнительно принудительно сбрасывает буфер вывода.

## Как оформить ответ для автоматической проверки

Ввод и вывод должны соответствовать условию задачи:

- выводите только ответ, без приглашений «Введите число» и подписей «Результат»;
- сохраняйте требуемый порядок значений, регистр букв, пробелы и переводы строк;
- для вещественных чисел задавайте требуемое количество знаков и учитывайте погрешность, указанную в задаче;

- не заменяйте целочисленное деление вещественным или наоборот, если это меняет смысл ответа;
- диагностические сообщения не включайте в стандартный вывод.

Для диагностики в C можно использовать `fprintf(stderr, "Input error\n");`, а в C++ — `std::cerr << "Input error\n";`. `stderr` и `std::cerr` относятся к стандартному потоку ошибок. Возврат `1` из `main` сообщает о неуспешном завершении программы, но сам по себе текст не выводит.

Вывод может накапливаться в буфере. Если интерактивная программа должна показать приглашение без перевода строки до ожидания ввода, после `printf` вызовите `fflush(stdout)`. Для задач на сайте приглашения не требуются. Не используйте `fflush(stdin)` для удаления введённых символов: стандарт C не задаёт такую операцию для входного потока.

## 1.7. Контрольные вопросы

1. Чем объявление переменной отличается от присваивания ей нового значения? Что такое инициализация?
2. Для чего используются фигурные скобки и точка с запятой? Почему точка с запятой после условия может изменить поведение программы?
3. Можно ли по имени типа int определить его точный размер на любой платформе? Как выяснить размер и границы значений типа?
4. Чем отличаются знаковые и беззнаковые целые типы? Почему беззнаковый тип не защищает от ошибок вычисления?
5. Почему не следует читать локальную переменную, которой ещё не присвоено значение?
6. Какой результат даст выражение 7 / 2, если оба операнда имеют тип int? Что изменится при записи 7 / 2.0?
7. Почему присваивание результата целочисленного деления переменной double не восстанавливает дробную часть?
8. Чем отличаются = и ==? Какие значения считаются истинными и ложными в условии C?
9. Как короткое замыкание позволяет безопасно проверить делитель перед выполнением деления?
10. Почему скобки в выражении не определяют порядок вычисления всех его операндов?
11. Чем отличается переполнение знакового целого от арифметики беззнаковых целых?
12. Как выбирается ветвь в последовательности if / else if / else? Когда удобнее использовать switch?
13. Что произойдёт в switch, если после кода case отсутствует break?
14. Чем отличаются while, do while и for? Какой из этих циклов гарантирует хотя бы одно выполнение тела?
15. Для цикла for (int i = 0; i < 5; ++i) укажите значения i, при которых выполняется тело, и количество итераций.
16. Чем отличаются break и continue? Как они действуют при вложенных циклах?
17. Как связаны объявление, определение и вызов функции? Зачем компилятору знать объявление до вызова?
18. Что означает void в возвращаемом типе функции и в списке её параметров?
19. Почему изменение обычного параметра функции не изменяет переменную, пере-

данную при вызове?

20. Как выбрать параметры и возвращаемое значение функции, вычисляющей минимум двух чисел? Какие случаи следует проверить?
21. Почему содержательные имена и отступы важны даже для кода, который успешно компилируется? Какие решения остаются за программистом при использовании ИИ?
22. Как return влияет на выполнение функции? Почему выход из main на компьютере и завершение main в проекте микроконтроллера нельзя автоматически считать одинаковыми действиями?
23. Чем аргументы printf отличаются от аргументов scanf? Для чего при вводе записывается & перед именем переменной?
24. Почему для чтения double нужен %lf, а для вывода — %f? Что задаёт формат %.6f?
25. Как определить, удалось ли прочитать все ожидаемые значения? Почему числовой ввод обычно не зависит от разделения чисел по строкам?
26. Почему для чтения символа после числа часто используют формат "%c"?
27. Почему сохранение результата в long long не исправляет переполнение произведения, уже вычисленного в int?
28. Как прочитать и вывести число средствами C++? Что задаёт сочетание std::fixed и std::setprecision(6)?
29. Какие сообщения и разделители должны присутствовать в выводе программы, отправляемой на автоматическую проверку?

# Глава 2. Работа с данными и составными типами

В первой главе функции получали отдельные значения. Теперь рассмотрим доступ к исходным объектам через указатели, обработку массивов, объединение данных в структуры и представление состояний перечислениями и битовыми флагами.

В примерах продолжается задача обработки температур. Диапазон допустимых входных значений и порог тревоги задаются условиями этой учебной задачи. Фрагменты C размещайте внутри main; директивы #include и определения типов — перед main, если не указано другое окружение. Правила C для указателей, массивов, структур и перечислений описаны в N1570, §§6.3.2.1, 6.5.2–6.5.6, 6.7.2–6.7.3 и 6.7.6 [3]. Ссылки C++ рассматриваются отдельно в 2.2 [4].

## 2.1. Адрес объекта и указатели

**Адрес объекта** определяет его положение в памяти. Унарный оператор `&` получает адрес объекта. **Указатель** — объект, значение которого может указывать на другой объект. Тип указателя определяет тип объекта, к которому через него обращаются.

```c
int main(void)
{
    int raw = 240;
    int *pointer = &raw;

    int saved = *pointer;
    *pointer = 250;

    return 0;
}
```

Объявление `int *pointer` создаёт переменную типа «указатель на `int`». Выражение `&raw` имеет этот тип и указывает на `raw`. После инициализации `pointer` хранит адрес `raw`, а не значение `240`.

Унарный оператор `*` выполняет **разыменование**: выражение `*pointer` обозначает объект, на который указывает `pointer`. В `int saved = *pointer;` читается значение `raw`. В `*pointer = 250;` изменяется сам объект `raw`. Переменная `saved` сохраняет ранее скопированное значение `240`.

Звёздочка в объявлении описывает тип, а звёздочка в выражении выполняет операцию. Их назначение различается. В записи `int *a, b;` только `a` является указателем; `b` имеет тип `int`. Для ясности указатели далее объявляются отдельно.

Разыменование требует, чтобы указатель указывал на существующий объект подходящего типа, доступный для соответствующего чтения или записи. Объявление `int *pointer;` само по себе не создаёт объект `int` и не задаёт допустимый адрес. Читать такой неинициализированный указатель и обращаться через него к памяти нельзя.

### Нулевой указатель

**Нулевой указатель** не указывает ни на один объект. Для его записи стандартные заголовки, в частности `<stddef.h>`, предоставляют макрос `NULL`.

```c
#include <stddef.h>

int main(void)
{
    int raw = 240;
    int *pointer = NULL;

    if (pointer == NULL) {
        pointer = &raw;
    }
    *pointer = 250;

    return 0;
}
```

Нулевой указатель можно хранить, присваивать и сравнивать. Разыменовывать его нельзя. Проверка `pointer != NULL` устанавливает только отсутствие нулевого значения: она не доказывает, что любой другой адрес пригоден для обращения. Действительность указателя обеспечивается способом его получения и временем существования объекта.

## Выходной результат через указатель

Функция может возвращать статус выполнения обычным возвращаемым значением, а вычисленные данные записывать в объект вызывающей функции через указатель. Такой параметр называют **выходным параметром**.

```c
#include <stddef.h>

int decode_temperature(int raw, double *output)
{
    if (output == NULL || raw < 0 || raw > 500) {
        return 0;
    }

    *output = raw / 10.0;
    return 1;
}

int main(void)
{
    double temperature = 0.0;
    int status = decode_temperature(280, &temperature);

    if (status == 0) {
        return 1;
    }

    int alarm = temperature >= 27.0;
    return 0;
}
```

`raw` содержит температуру в десятых долях градуса: `280` соответствует `28.0`. Допустимый диапазон исходных значений — от `0` до `500` включительно. Функция возвращает `1` при успехе и `0` при ошибке. При ошибке она не изменяет выходной объект.

Вызов передаёт адрес `temperature`. Параметр `output` получает копию этого адреса. Выражение `*output = ...` записывает результат в `temperature`, поэтому после возврата

результат доступен вызывающей функции. Указатель должен указывать на доступный для записи объект `double`; передача произвольного ненулевого значения этому требованию не соответствует.

Статус проверяется до использования результата как успешно вычисленного значения. Начальная инициализация `temperature` не заменяет эту проверку: при ошибке сохранённое значение не является результатом данного вызова.

### Указатель тоже передаётся по значению

В C все параметры передаются по значению, включая указатели. Функция получает собственную переменную с копией переданного значения указателя. Изменение этой переменной и изменение объекта через неё — разные действия.

```
#include <stddef.h>

void clear_pointer(int *pointer)
{
    *pointer = 0;
    pointer = NULL;
}

int main(void)
{
    int raw = 240;
    int *pointer = &raw;

    clear_pointer(pointer);

    return 0;
}
```

Инструкция `*pointer = 0;` изменяет `raw`. Последующая инструкция `pointer = NULL;` меняет только локальный параметр функции. Указатель в `main` продолжает указывать на `raw`. Для этого вызова функция требует действительный указатель на доступный для записи `int`.

Адрес локального объекта нельзя использовать после завершения его существования. В частности, нельзя возвращать адрес обычной локальной переменной функции и затем разыменовывать его у вызывающей стороны: после выхода из функции такого объекта уже нет. Для выдачи результата в показанном примере используется объект вызывающей функции, существующий на протяжении всего вызова. Подробные правила времени жизни рассматриваются в главе 3.

## 2.2. Ссылки в C++

**Ссылка** — другое имя уже существующего объекта. В объявлении ссылки после типа ставится `&`; при создании ей сразу указывают объект, с которым она связана.

Примеры этой подглавы написаны на C++17. Сохраняйте их в файл `main.cpp` и собирайте командой `g++ -std=c++17 -Wall -Wextra -g -O0 main.cpp -o main.exe`.

```
int main()
{
    int raw_temperature = 253;
    int& reading = raw_temperature;

    reading = 260;
    int copy = reading;

    return 0;
}
```

Присваивание через `reading` меняет `raw_temperature`: обе записи обозначают один объект. Переменная `copy` получает копию значения 260 и остаётся отдельным объектом.

Знак `&` здесь относится к объявлению ссылки. В выражении `&raw_temperature` он получает адрес объекта, как и в C. Назначение символа определяется местом его использования.

## Передача по ссылке

Параметр-ссылка позволяет функции обращаться к исходному объекту по имени параметра. Для вызова передают сам объект; явно получать его адрес не требуется.

```
void apply_correction(int& raw_temperature, int correction)
{
    raw_temperature += correction;
}

int main()
{
    int raw_temperature = 253;
    apply_correction(raw_temperature, -3);

    return 0;
}
```

После вызова `raw_temperature` равна 250. Параметр `correction` передан по значению, а `raw_temperature` — по ссылке. Для этой функции сумма должна быть представима в `int`, как и при обычном сложении.

В C такую задачу решают параметром-указателем: функция получает адрес и использует разыменование. В C++ параметр-ссылка предоставляет прямой доступ к объекту через имя параметра. При чтении вызова `apply_correction(raw_temperature, -3)` нужно учитывать объявление функции: сам синтаксис вызова не показывает, будет ли изменён аргумент.

## Связь ссылки с объектом

Ссылку нельзя переназначить другому объекту. Присваивание меняет связанный объект:

```
int main()
{
    int first = 24;
    int second = 30;
    int& reading = first;

    reading = second;
    second = 35;

    return 0;
}
```

В результате `first` равна 30, `second` — 35. Ссылка `reading` по-прежнему обозначает `first`, а не `second`.

У ссылки нет допустимого «нулевого» состояния, аналогичного нулевому указателю. Для создания и использования такой ссылки нужен существующий объект, который продолжает существовать при обращениях через неё. Возвращать ссылку на обычную локальную переменную функции нельзя: после выхода из функции эта переменная перестаёт существовать. Указатель удобен, когда отсутствие объекта или смена адресуемого объекта являются частью задачи; ссылка — когда функция должна работать с переданным объектом.

Здесь рассмотрены ссылки вида `T&`. Другие виды ссылок относятся к последующим темам C++.

## 2.3. Массивы: элементы, границы и передача в функции

**Массив** — последовательность элементов одного типа, размещённых непрерывно в памяти. Число элементов определяет размер массива. Доступ к элементу выполняется по индексу, отсчитываемому от нуля.

```
int main(void)
{
    int raw[5] = {240, 250, 260, -1, 280};

    int first = raw[0];
    raw[3] = 270;

    return 0;
}
```

`raw` содержит пять объектов `int`. Допустимые индексы — `0`, `1`, `2`, `3`, `4`. Выражение `raw[0]` обозначает первый элемент, а `raw[3]` — четвёртый. Значение `-1` в исходном наборе можно использовать как признак недопустимого измерения: это значение элемента, а не отрицательный индекс.

Инициализаторы в фигурных скобках задают начальные значения элементов по порядку. Если их меньше, чем элементов, оставшиеся элементы инициализируются нулями. Например, `int values[4] = {10, 20};` создаёт значения `10`, `20`, `0`, `0`. Запись `int values[4] = {0};` инициализирует весь массив нулями. Без инициализации обычный локальный массив не получает автоматически определённых начальных значений.

При наличии списка инициализаторов размер можно не писать: `int raw[] = {240, 250, 260, -1, 280};` создаёт массив из пяти элементов. Размер такого массива после

создания не изменяется. Присвоить весь массив другому массиву оператором `=` нельзя; отдельные элементы можно читать и изменять.

## Границы и размер массива

Для массива из `count` элементов индекс должен удовлетворять условию `0 <= index < count`. Язык не предусматривает обязательной проверки границ при каждом обращении. Чтение `raw[5]` в предыдущем примере выходит за пределы массива и вызывает неопределённое поведение. Отсутствие сообщения об ошибке не означает допустимость такого обращения.

Оператор `sizeof` для самого массива возвращает размер всех его элементов в байтах. Число элементов получают делением размера массива на размер одного элемента:

```
#include <stddef.h>

int main(void)
{
    int raw[] = {240, 250, 260, -1, 280};
    size_t count = sizeof raw / sizeof raw[0];
    int invalid_count = 0;

    for (size_t i = 0; i < count; ++i) {
        if (raw[i] < 0 || raw[i] > 500) {
            ++invalid_count;
        }
    }

    return 0;
}
```

`size_t` — беззнаковый целочисленный тип результата `sizeof`. Здесь он используется также для числа элементов и индекса. Условие `i < count` обеспечивает обращение только к существующим элементам. Значение `count` не является последним допустимым индексом: последний индекс при непустом массиве равен `count - 1`.

## Преобразование массива в указатель

В большинстве выражений имя массива преобразуется в указатель на его первый элемент. Например, после `int *pointer = raw;` указатель имеет значение `&raw[0]`. При этом сам массив остаётся массивом: указатель и массив — разные объекты и разные типы.

Операнд `sizeof` не подвергается такому преобразованию. Поэтому `sizeof raw` возвращает размер массива, а `sizeof pointer` — размер переменной-указателя. Формула `sizeof pointer / sizeof pointer[0]` не определяет число элементов исходного массива.

В списке параметров функции запись `int raw[]` корректируется языком до `int *raw`. Ни вариант с пустыми скобками, ни запись `int raw[5]` не передают размер массива и не создают его копию. Для обработки массива функция обычно получает отдельно указатель на первый элемент и число элементов.

## Передача массива и вычисление среднего

Следующая функция вычисляет среднюю температуру по допустимым измерениям. Недопустимые исходные значения пропускаются. Успех означает наличие хотя бы одного допустимого измерения; при ошибке выходное значение сохраняется.

```c
#include <stddef.h>

int average_temperature(int *raw, size_t count, double *output)
{
    if (raw == NULL || output == NULL || count == 0) {
        return 0;
    }

    double sum = 0.0;
    size_t valid_count = 0;

    for (size_t i = 0; i < count; ++i) {
        if (raw[i] >= 0 && raw[i] <= 500) {
            sum += raw[i] / 10.0;
            ++valid_count;
        }
    }

    if (valid_count == 0) {
        return 0;
    }

    *output = sum / valid_count;
    return 1;
}

int main(void)
{
    int raw[] = {240, 250, 260, -1, 280};
    size_t count = sizeof raw / sizeof raw[0];
    double average = 0.0;

    if (average_temperature(raw, count, &average) == 0) {
        return 1;
    }

    int alarm = average >= 27.0;
    return 0;
}
```

Среднее вычисляется для температур `24.0`, `25.0`, `26.0` и `28.0` и равно `25.75`. Значение `-1` пропущено. Число допустимых измерений учитывается отдельно от полного размера массива, поэтому делитель равен четырём, а не пяти. Проверка `valid_count == 0` предотвращает деление на ноль.

Вызывающая сторона обязана передать указатель на последовательность как минимум из `count` доступных элементов `int`. По одному указателю функция не может проверить действительную длину этой последовательности. Внутри `average_temperature` выражение `sizeof raw` дало бы размер указателя, а не исходного массива.

## Арифметика указателей

При сложении указателя с целым числом смещение измеряется в элементах указанного типа. Если `pointer` указывает на первый `int` массива, то `pointer + 1` указывает на следующий `int`. Выражения `raw[i]` и `*(raw + i)` обозначают один элемент.

```
int main(void)
{
    int raw[] = {240, 250, 260};
    int *pointer = raw;
    int *end = raw + 3;
    int total = 0;

    while (pointer != end) {
        total += *pointer;
        ++pointer;
    }

    return 0;
}
```

Допускается получать указатели на элементы одного массива и на позицию непосредственно после последнего элемента, называемую **one-past**. Здесь `end` указывает на такую позицию. Этот указатель используется как граница, но не разыменовывается. Получать результат арифметики за пределами этих позиций нельзя. После обработки последнего элемента `pointer` становится равным `end`, и цикл завершается до следующего разыменования.

## Строки как массивы символов

**Строка C** — последовательность элементов char, завершающаяся нулевым символом `\0`. Например, `char label[] = "OK";` создаёт массив из трёх элементов: `'O'`, `'K'`, `'\0'`. Последний элемент отмечает конец строки и не является видимой буквой. Символ `'0'` представляет цифру и отличается от `'\0'`.

Массив `char` без завершающего нулевого символа сам по себе не является строкой C. Размер массива со строкой включает место для этого символа; длина текста его не включает. Строка `"ABC"` занимает четыре элемента char: три для букв и один для завершающего нуля. В многобайтной кодировке, например UTF-8, один видимый символ может занимать несколько элементов char.

## 2.4. Структуры и передача составных данных

**Структура** — тип данных, объединяющий несколько именованных объектов в одно составное значение. Составляющие структуры называются **членами** или **полями**. Поля могут иметь разные типы; каждое имеет собственное имя. Структура позволяет представить связанные данные одним объектом: например, измерение вместе с признаком его достоверности.

Определение типа структуры имеет следующий вид:

```
struct Measurement {
    int raw;
    int valid;
};
```

Здесь `Measurement` — имя структуры, `raw` и `valid` — её поля. Тип объекта в C записывается как `struct Measurement`. Определение задаёт состав типа, но само по себе не создаёт переменную. Точка с запятой после закрывающей фигурной скобки обязательна.

В примерах этой главы температура хранится в десятых долях градуса: `240` означает `24,0 °C`. Допустимые значения — от `0` до `500` включительно. Значение вне этого диапазона считается ошибочным измерением. Тревога соответствует достоверной температуре не ниже `27,0 °C`, то есть `raw >= 270`.

Следующий фрагмент требует приведённого выше определения `struct Measurement` и размещается внутри функции:

```
struct Measurement sample = {240, 1};
int temperature = sample.raw;
sample.valid = 0;
```

Для обращения к полю объекта используется оператор `.`: слева записывается объект структуры, справа — имя поля. Поле можно читать и изменять по тем же правилам, что и отдельную переменную соответствующего типа. После фрагмента `temperature` равна `240`, а поля `sample` равны `240` и `0`.

## Инициализация и копирование

В инициализаторе `{240, 1}` значения соответствуют порядку объявления полей. В C11 можно указать имена полей явно:

```
/* Фрагмент внутри функции; требуется определение struct Measurement. */
struct Measurement sample = {
    .valid = 1,
    .raw = 240
};
```

Такая запись называется инициализацией с обозначениями. Порядок записанных полей здесь не обязан совпадать с порядком их объявления. Она помогает понять назначение значений без обращения к определению структуры.

Если при инициализации некоторые поля не указаны, они получают нулевые значения в соответствии со своими типами. Для структуры с двумя целочисленными полями запись `struct Measurement sample = {0};` инициализирует оба поля нулями. Однако объявление `struct Measurement sample;` внутри функции без инициализатора не задаёт начальные значения полей: читать их до присваивания нельзя.

Объект структуры можно присвоить другому объекту совместимого структурного типа:

```
/* Фрагмент внутри функции; требуется определение struct Measurement. */
struct Measurement first = {.raw = 240, .valid = 1};
struct Measurement second = first;
second = first;
second.raw = 250;
```

Инициализация `second = first` и последующее присваивание копируют значения полей. После изменения `second.raw` поле `first.raw` остаётся равным 240: это два разных объекта. Если структура содержит массив, его элементы также копируются при присваивании всей структуры, хотя присваивать отдельные массивы оператором = нельзя.

Копирование структуры с полями-указателями копирует сами указатели, а не объекты, на которые они указывают. Поэтому два полученных указателя могут обращаться к одному и тому же объекту. При выборе способа передачи данных нужно учитывать состав структуры.

## Передача по значению и через указатель

Структура может быть параметром функции и возвращаемым значением. При передаче по значению параметр является отдельным объектом, и изменение его полей не изменяет структуру вызывающего кода.

Полная программа C11:

```
struct Measurement {
    int raw;
    int valid;
};

struct Measurement validate(struct Measurement sample)
{
    sample.valid = sample.raw >= 0 && sample.raw <= 500;
    return sample;
}

int main(void)
{
    struct Measurement original = {.raw = 240, .valid = 0};
    struct Measurement checked = validate(original);

    /* original.valid == 0; checked.valid == 1. */
    return 0;
}
```

Функция получает копию значения `original`, вычисляет признак достоверности и возвращает структуру. Результат сохраняется в `checked`. Здесь возвращается составное значение, а не адрес локального объекта; такой способ возврата корректен.

Если функция должна изменить исходную структуру, ей передают указатель. Для доступа к полю через указатель используется оператор `->`. Запись `sample->valid` означает то же обращение к полю, что и `(*sample).valid`.

Полная программа C11:

```
struct Measurement {
    int raw;
    int valid;
};

void validate(struct Measurement *sample)
{
    sample->valid = sample->raw >= 0 && sample->raw <= 500;
}

int main(void)
{
    struct Measurement sample = {.raw = -1, .valid = 1};
    validate(&sample);

    /* Теперь sample.valid == 0. */
    return 0;
}
```

Значение указателя также передаётся по значению, но обращение через него позволяет изменить исходный объект. В этой функции требуется указатель на существующий доступный для изменения объект `struct Measurement`; нулевой указатель передавать нельзя. Это требование является частью условий использования функции.

Передача по значению удобна для получения независимого набора данных и возврата результата. Указатель нужен для изменения объекта вызывающего кода; его также применяют для доступа к большим структурам без передачи всего значения. Выбор способа передачи должен прежде всего выражать назначение функции.

## Имена типов с `typedef`

`typedef` вводит дополнительное имя уже существующего типа. Это не создание нового отдельного типа и не объявление переменной.

```
struct Measurement {
    int raw;
    int valid;
};

typedef struct Measurement Measurement;
```

После этих объявлений записи `Measurement sample;` и `struct Measurement sample;` задают объекты одного типа. Определение структуры и создание имени можно объединить:

```
typedef struct Measurement {
    int raw;
    int valid;
} Measurement;
```

Две приведённые формы являются альтернативами: их не нужно записывать вместе. Далее дополнительное имя будет применено к структуре результатов обработки.

### Размещение полей в памяти

Поля структуры размещаются в порядке объявления, но между ними и после последнего поля могут находиться дополнительные байты для выравнивания — **padding**. Поэтому размер структуры не обязан равняться сумме размеров её полей.

Набор полей описывает логические данные, но не универсальный формат сообщения или файла. Передавать сырые байты структуры как переносимый формат нельзя без отдельно согласованного представления: влияют выравнивание, размеры типов и порядок байтов. Для обмена данные обычно записывают в определённом формате по отдельным полям.

## 2.5. Перечисления

**Перечисление** — целочисленный тип с набором именованных констант. Оно позволяет записывать состояния и варианты результата понятными именами вместо чисел.

```
enum Result {
    RESULT_OK,
    RESULT_NO_DATA,
    RESULT_BAD_ARGUMENT
};
```

В C11 перечисленные имена констант имеют тип `int`. Если значение первой константы не задано явно, оно равно `0`. Каждая следующая константа без явного значения получает предыдущее значение плюс один. В примере значения равны `0`, `1` и `2`.

Значения можно задавать явно:

```
enum Mode {
    MODE_IDLE = 0,
    MODE_MEASURE = 10,
    MODE_REPORT,
    MODE_STOP = 20
};
```

Здесь `MODE_REPORT` равна `11`. Значения должны быть целочисленными константными выражениями, представимыми типом `int`. У разных констант могут быть одинаковые значения. Имена констант записываются без приставки имени типа; написание заглавными буквами — соглашение об оформлении, а не требование языка.

Тип переменной в C записывается с ключевым словом `enum`. Следующий фрагмент требует определения `enum Result` и размещается внутри функции:

```
enum Result status = RESULT_NO_DATA;

if (status == RESULT_NO_DATA) {
    status = RESULT_OK;
}
```

Перечисление удобно использовать и в `switch`. Однако в C оно не обеспечивает автоматическую проверку принадлежности значения объявленному набору. Например, присваивание `status = 7;` не становится запрещённым только потому, что константа с таким значением отсутствует. Проверку значения, поступившего извне, необходимо выполнять явно.

Для самого перечислимого типа реализация выбирает совместимый целочисленный тип. Поэтому не следует считать, что размер любого `enum` всегда равен размеру `int`, хотя именованные константы перечисления в C11 имеют именно тип `int`.

## Объединяющий пример: результат обработки измерений

Соберём результат обработки массива в структуру: число допустимых измерений, число срабатываний по порогу, средняя температура в градусах Цельсия и статус. Статус различает успешный расчёт, отсутствие данных и ошибку аргументов.

```
#include <stddef.h>

enum Result {
    RESULT_OK,
    RESULT_NO_DATA,
    RESULT_BAD_ARGUMENT
};

typedef struct Statistics {
    size_t valid_count;
    size_t alarm_count;
    double average_c;
    enum Result status;
} Statistics;

Statistics analyse(int *raw, size_t count)
{
    Statistics result = {
        .valid_count = 0,
        .alarm_count = 0,
        .average_c = 0.0,
        .status = RESULT_NO_DATA
    };
    double sum_c = 0.0;

    if (count > 0 && raw == NULL) {
        result.status = RESULT_BAD_ARGUMENT;
        return result;
    }

    for (size_t i = 0; i < count; ++i) {
        if (raw[i] < 0 || raw[i] > 500) {
            continue;
        }

        ++result.valid_count;
        sum_c += raw[i] / 10.0;

        if (raw[i] >= 270) {
            ++result.alarm_count;
        }
    }

    if (result.valid_count > 0) {
        result.average_c = sum_c / result.valid_count;
        result.status = RESULT_OK;
    }
```

```
    return result;
}

int main(void)
{
    int raw[] = {240, 250, 260, -1, 280};
    size_t count = sizeof raw / sizeof raw[0];
    Statistics result = analyse(raw, count);

    if (result.status != RESULT_OK) {
        return 1;
    }

    /* Здесь результат доступен для дальнейшей обработки. */
    return 0;
}
```

В массиве четыре допустимых измерения. Одно из них достигает порога 27 °C; средняя температура равна 25,75 °C. Функция возвращает самостоятельную структуру результата и не изменяет исходные измерения.

При нулевом count или отсутствии допустимых измерений статус равен RESULT_NO_DATA. Поле average_c в этом случае не содержит вычисленного среднего. Его нулевое значение само по себе не обозначает отсутствие данных: температура 0 °C допустима, поэтому вызывающий код проверяет status.

При положительном count вызывающий код обязан передать массив как минимум из count доступных элементов. Проверка NULL выявляет отсутствие адреса, но не проверяет длину массива.

## 2.6. Двоичное представление целых чисел и побитовые операции

### Биты и запись чисел

Беззнаковое целое число можно рассматривать как последовательность битов. Каждый бит равен `0` или `1` и имеет вес, зависящий от позиции. Нумерация начинается с **бита 0**, младшего бита, имеющего вес `2^0 = 1`. Бит 1 имеет вес `2`, бит 2 — `4`, бит 3 — `8`. Значение числа равно сумме весов установленных, то есть равных единице, битов.

Например, число `13` представляется последовательностью `1101`: установлены биты 3, 2 и 0, поэтому `13 = 8 + 4 + 1`.

Тип `uint32_t` из `<stdint.h>` содержит ровно 32 бита значения и представляет числа от `0` до `2^32 - 1`. Для работы с битами далее используются беззнаковые значения этого типа.

Шестнадцатеричная запись начинается с `0x`. Одна шестнадцатеричная цифра соответствует четырём битам: `0x1` соответствует `0001`, `0xA` — `1010`, `0xF` — `1111`. Таким образом, `0x0D` и десятичное `13` обозначают одно значение. Начальные нули после `0x` служат для наглядности и не меняют значение.

Суффикс `u` обозначает беззнаковую целочисленную константу. Он не задаёт фиксированную разрядность: тип литерала выбирается из подходящих беззнаковых типов.

Например, `0x1u` имеет тип `unsigned int`.

## Побитовые операторы

Побитовые операторы обрабатывают соответствующие биты целочисленных операндов независимо друг от друга:

| Оператор | Действие над одним битом |
|---|---|
| `&` | И: результат равен `1`, если оба бита равны `1` |
| `\|` | ИЛИ: результат равен `1`, если хотя бы один бит равен `1` |
| `^` | Исключающее ИЛИ: результат равен `1`, если биты различаются |
| `~` | Инверсия: заменяет `0` на `1` и `1` на `0` |

Таблица 6: Побитовые операторы

```
#include <stdint.h>

/* Фрагмент тела функции. */
uint32_t a = 0x0Cu;       /* младшие четыре бита: 1100 */
uint32_t b = 0x0Au;       /* младшие четыре бита: 1010 */
uint32_t both = a & b;    /* 0x08: 1000 */
uint32_t either = a | b;  /* 0x0E: 1110 */
uint32_t different = a ^ b; /* 0x06: 0110 */
uint32_t inverted = ~a;   /* 0xFFFFFFF3 */
```

Инверсия затрагивает все биты типа вычисления, а не только разряды, явно записанные в литерале. Побитовые `&` и `|` отличаются от логических `&&` и `||`: они не сводят число к признаку истинности и не обеспечивают короткое замыкание.

Перед побитовыми операциями действуют целочисленные продвижения и преобразования из раздела 1.3. В частности, `~` для переменной `unsigned char` обычно вычисляется в `int`. Использование `uint32_t` задаёт ширину хранимого результата, а тип вычисления определяется правилами продвижения и преобразования операндов.

## Сдвиги

Оператор `value << n` сдвигает биты влево на `n` позиций, а `value >> n` — вправо. Для беззнакового операнда освобождённые позиции заполняются нулями. Биты, вышедшие за границы типа вычисления, теряются.

```
#include <stdint.h>

/* Фрагмент тела функции. */
uint32_t value = 0x0Du;   /* 1101 */
uint32_t left = value << 2;  /* 0x34: 110100 */
uint32_t right = value >> 2; /* 0x03: 11 */
uint32_t one = 0x1u;
uint32_t bit3_mask = one << 3; /* 0x08: установлен только бит 3 */
```

Число позиций должно быть неотрицательным и строго меньше ширины продвинутого левого операнда. Для 32-разрядного типа вычисления это диапазон от `0` до `31`. Сдвиг на `32` в этом случае, как и отрицательный сдвиг, имеет неопределённое поведение. Тип переменной назначения это ограничение не изменяет.

Для битовых вычислений следует использовать беззнаковый левый операнд. Правый сдвиг отрицательного знакового числа в C11 зависит от реализации; левый сдвиг знакового числа имеет дополнительные ограничения на допустимые значения и результат.

Запись `1u << n` не является универсальным способом получить 32-битную маску: ширина `unsigned int` зависит от реализации. Проверять нужно ширину фактического типа вычисления, а не только переменной, в которую сохраняется результат.

## Маски и флаги

**Маска** — число, установленными битами которого отмечены интересующие позиции. Если каждому признаку назначить отдельный бит, несколько логических признаков можно хранить в одном целом числе. Такие признаки называются **флагами**.

Код перечисления обозначает один выбранный вариант: например, RESULT_BAD_ARGUMENT имеет значение 2. Битовая маска объединяет независимые признаки, которые могут присутствовать одновременно. Пусть бит 0 означает наличие достоверных температурных данных, бит 1 — перегрев, бит 2 — ошибку датчика. Остальные биты пока не используются.

```
#include <stdint.h>

/* Фрагмент тела функции. */
uint32_t valid_mask = 0x1u;
uint32_t overheat_mask = 0x2u;
uint32_t sensor_error_mask = 0x4u;
uint32_t flags = 0;

flags |= valid_mask;           /* установить бит достоверности */
flags |= overheat_mask;        /* установить бит перегрева: flags == 3 */
flags &= ~overheat_mask;       /* сбросить бит перегрева: flags == 1 */
flags ^= sensor_error_mask;    /* переключить бит ошибки: flags == 5 */
flags ^= sensor_error_mask;    /* переключить повторно: flags == 1 */
```

Операция `| mask` устанавливает отмеченные биты, `& ~mask` сбрасывает их, `^ mask` переключает. Биты вне маски сохраняются. Присваивание `flags = valid_mask` имеет другой смысл: заменяет всё значение и уничтожает остальные флаги.

Для проверки бита нужно выделить его с помощью `&`, затем сравнить результат с нулём. Маска может включать несколько битов; проверки «установлен хотя бы один» и «установлены все» различаются.

```
#include <stdint.h>

/* Фрагмент тела функции. */
uint32_t flags = 0x3u;
uint32_t valid_mask = 0x1u;
uint32_t error_mask = 0x6u; /* перегрев или ошибка датчика */
uint32_t required_mask = 0x3u;

int valid = (flags & valid_mask) != 0;            /* 1 */
int any_error = (flags & error_mask) != 0;        /* 1 */
int all_required = (flags & required_mask) == required_mask; /* 1 */
```

Скобки вокруг побитового выражения необходимы для выбранного смысла: операторы сравнения имеют более высокий приоритет, чем `&`. Например, `flags & valid_mask`

`!= 0` разбирается как `flags & (valid_mask != 0)`.

### Битовые поля внутри числа

Группа соседних битов может хранить небольшое числовое значение. Например, биты 4–6 образуют поле шириной три бита, способное представлять значения от `0` до `7`. Для извлечения поля сначала выделяют его маской, затем сдвигают к младшим позициям.

```
#include <stdint.h>

/* Фрагмент тела функции. */
uint32_t packed = 0x53u; /* поле равно 5, младшие флаги равны 3 */
uint32_t field_mask = 0x70u;
uint32_t field = (packed & field_mask) >> 4; /* 5 */

uint32_t new_field = 2; /* допустимые значения: 0..7 */
packed = (packed & ~field_mask) | ((new_field << 4) & field_mask);
/* packed равно 0x23: поле заменено, остальные биты сохранены */
```

При записи сначала очищается старое поле, затем добавляются новые биты. Простое `packed |= new_field << 4` не заменяет прежнее значение: единицы старого поля могут сохраниться. Ограничение маской предотвращает изменение соседних полей, но не заменяет проверку диапазона `new_field`: слишком большое значение будет усечено, а не автоматически признано ошибкой.

Рассмотренное поле является группой битов обычного целого числа. Это не отдельное поле структуры и не специальное объявление битового поля языка C.

## 2.7. Контрольные вопросы

1. Чем отличаются значение переменной, её адрес и значение указателя на неё?
2. Объясните назначение `&` и `*` в объявлениях и выражениях. Как изменится исходная переменная после присваивания через указатель?
3. Что обозначает нулевой указатель? Почему проверки на NULL недостаточно, чтобы доказать допустимость любого ненулевого адреса?
4. Указатель передаётся в функцию по значению. Почему функция при этом может изменить объект вызывающего кода?
5. Что произойдёт, если функция присвоит своему параметру-указателю другой адрес? Изменится ли указатель в вызывающем коде?
6. Почему нельзя вернуть указатель или ссылку на обычную локальную переменную функции?
7. Как функция может вернуть статус выполнения и отдельно записать вычисленный результат? Когда выходной результат разрешено использовать?
8. Чем отличается параметр-ссылка C++ от параметра-указателя C? Как присваивание через ссылку влияет на объект?
9. Почему запись `reading = second` не переназначает ссылку reading?
10. Какие индексы допустимы для массива из пяти элементов? Что случится при обращении к шестому элементу?
11. Чем отличаются массив и указатель на его первый элемент? Почему sizeof массива и sizeof указателя могут давать разные результаты?
12. Почему размер массива передают в функцию отдельным параметром? Можно ли восстановить его по одному указателю?

13. Где можно применять арифметику указателей? Для чего допустим адрес сразу после массива и почему его нельзя разыменовать?
14. Чем строка C отличается от произвольного массива char? Сколько элементов char нужно для хранения строки ”ABC”вместе с завершающим нулём?
15. Какие данные удобно объединить в структуру? Чем отличаются операции `.` и `->`?
16. Как отличается передача структуры по значению от передачи указателя на неё? Что изменится в исходной структуре?
17. Для чего используется typedef? Создаёт ли он отдельный новый тип?
18. Почему размер структуры может превышать сумму размеров её членов? Почему копирование её байтов не задаёт переносимый формат сообщения?
19. Как перечисление помогает обозначать состояния? Гарантирует ли enum в C, что переменная содержит только перечисленные значения?
20. Почему номера состояний и независимые битовые флаги требуют разного представления?
21. Как установить, сбросить и переключить отдельный бит, сохранив остальные?
22. Чем отличается проверка хотя бы одного бита маски от проверки всех её битов?
23. Почему `&` и `&&` нельзя взаимозаменять? В каком случае правый операнд не вычисляется?
24. Какие ограничения действуют для числа позиций сдвига? Почему для битовых операций обычно выбирают беззнаковые типы?
25. Как выделить многобитное поле из целого числа и заменить его значение, сохранив другие поля?

# Глава 3. Объекты, время жизни и организация памяти

Чтобы выбирать способ передачи данных и хранить состояние программы, нужно понимать, где доступно имя объекта, сколько времени существует сам объект и какие операции над ним разрешены. В этой главе рассмотрим эти свойства, способы сохранения данных между вызовами и управление выделенной памятью.

Правила C для областей видимости и длительности хранения приведены в N1570, §§6.2.1–6.2.4, для квалификаторов — в §6.7.3, для управления динамической памятью — в §7.22.3 [3]. Примеры constexpr относятся к C++17 и выделены в отдельную подглаву.

## 3.1. Область видимости, время жизни и длительность хранения

**Область видимости имени** — часть исходного текста, в которой это имя обозначает объявленную сущность. **Время жизни объекта** — период выполнения программы, в течение которого объект существует и его хранилище сохраняется для него. **Длительность хранения** задаёт правило, по которому определяется время жизни объекта. Область видимости относится к тексту программы, а время жизни — к её выполнению.

### Область видимости блока

Имя переменной, объявленной внутри блока, имеет область видимости блока: оно доступно от места объявления до конца соответствующего блока. Во вложенном блоке можно обращаться к именам внешнего блока, если они не скрыты другим объявлением.

```
int main(void)
{
    int raw = 240;
    int saved = 0;

    {
        int correction = 10;
        raw += correction;
        saved = raw;
    }

    return saved != 250;
}
```

Внутри вложенного блока доступны `raw`, `saved` и `correction`. После закрывающей скобки имя `correction` недоступно, а `raw` и `saved` остаются доступны. Обращение к `correction` после этого блока было бы ошибкой компиляции.

Объявление во вложенном блоке может **скрывать** одноимённое объявление внешнего блока. При этом создаётся другой объект; внешний объект не исчезает и не изменяет своего имени в остальных частях программы.

```
int main(void)
{
    int raw = 240;
    int inner_value = 0;

    {
        int raw = 280;
        inner_value = raw;
    }

    return raw != 240 || inner_value != 280;
}
```

Внутри вложенного блока имя `raw` обозначает внутреннюю переменную со значением `280`. После выхода из блока оно снова обозначает внешнюю переменную со значением `240`. Такое скрытие допустимо, но затрудняет чтение: для разных данных предпочтительны разные имена.

Параметры функции также доступны в её теле. При разных вызовах одной функции создаются отдельные экземпляры её параметров и обычных локальных переменных. Совпадение имён локальных переменных разных функций не связывает их объекты.

## Область видимости файла

Объявление вне всех функций вводит имя с областью видимости файла. Оно доступно от места объявления до конца единицы трансляции — исходного файла после обработки директив включения. В повседневном описании переменные, объявленные таким образом, называют глобальными.

```
int alarm_limit_raw = 270;

int alarm_required(int raw)
{
    return raw >= alarm_limit_raw;
}

int main(void)
{
    int first_alarm = alarm_required(260);
    alarm_limit_raw = 280;
    int second_alarm = alarm_required(275);

    return first_alarm != 0 || second_alarm != 0;
}
```

Функции `alarm_required` и `main` используют один объект `alarm_limit_raw`. После изменения порога в `main` следующий вызов функции читает новое значение. Это отличается от параметра: каждый вызов получает собственный параметр `raw`, но глобальная переменная остаётся общей.

Область видимости файла не означает автоматической доступности имени в тексте другого исходного файла. Для многофайловой программы нужны соответствующие объявления и правила связывания имён; они рассматриваются отдельно. Для небольших функций передача данных параметрами делает зависимости явными: читатель видит, какие значения требуются функции.

## Автоматическая длительность хранения

Обычные локальные переменные и параметры функций имеют **автоматическую длительность хранения**. Для локального объекта фиксированного размера время жизни начинается при входе в связанный с ним блок и заканчивается при выходе из него. Само имя становится доступным только после его объявления. При новом входе в блок создаётся новый экземпляр объекта.

```
void invalidate_sample(int *output)
{
    int invalid_raw = -1;
    *output = invalid_raw;
}

int main(void)
{
    int raw = 240;
    invalidate_sample(&raw);
    return raw != -1;
}
```

`invalid_raw` существует во время выполнения блока функции. Функция копирует его значение в объект `raw`, который принадлежит вызывающей функции и продолжает существовать после возврата. Поэтому результат доступен в `main`.

Возвращать адрес обычной локальной переменной для дальнейшего использования нельзя: после выхода из функции объект прекращает существование. То же относится к адресу объекта вложенного блока после выхода из этого блока. Указатель не продлевает время жизни объекта. Указатель на завершивший существование объект называют **висячим указателем**; пользоваться им как указателем на доступный объект нельзя. В C++ ссылка также не обеспечивает продолжение жизни обычного локального объекта после выхода из его блока.

Область видимости имени и доступность объекта через указатель могут различаться. Функция `invalidate_sample` не видит локальное имя `raw` из `main`, но может изменить соответствующий объект через переданный адрес. Такой доступ допустим, поскольку объект `raw` продолжает существовать на протяжении вызова.

## Другие длительности хранения

Объекты со **статической длительностью хранения** существуют в течение всего выполнения программы. Эту длительность имеют, в частности, переменные, объявленные вне функций. Для них время жизни значительно шире области видимости имени: объект уже существует и тогда, когда выполнение находится в функции, где его имя недоступно. До начала выполнения программы такие объекты инициализируются; при отсутствии явного инициализатора арифметические объекты получают нулевое значение, а указатели — нулевое значение указателя.

**Выделенная длительность хранения** относится к объектам, хранилище для которых программа получает и освобождает явно с помощью средств управления динамической памятью. Объект может существовать независимо от завершения функции, которая получила память. Работа с этим хранилищем рассматривается в разделе о памяти. C11 также предусматривает длительность хранения потока для многопоточных программ; в заданиях этой работы она не используется.

Длительность хранения — понятие языка. Автоматический объект не следует определять как «переменную на стеке», а статический — как «переменную во Flash»: конкретное размещение определяется реализацией, средствами сборки и целевой платформой.

## 3.2. Ограничение изменения объектов: const

Квалификатор типа `const` ограничивает изменение объекта через выражения соответствующего типа. Он позволяет указать неизменяемые данные и описать доступ функции к объектам, получаемым через указатели.

### Объект типа const

```
int main(void)
{
    const int alarm_limit_raw = 270;
    int raw = 280;
    int alarm = raw >= alarm_limit_raw;

    return alarm != 1;
}
```

`alarm_limit_raw` имеет тип `const int`. Его значение задаётся при инициализации; последующее присваивание этому объекту недопустимо. Чтение значения и использование в выражениях разрешены. Объявление `int const alarm_limit_raw = 270;` означает то же самое.

`const` не означает определённое размещение в памяти и не гарантирует, что объект отсутствует в памяти программы. В C объект `const int` также не становится автоматически целочисленным константным выражением: например, его имя нельзя использовать как значение метки `case`. В C++17 правила использования некоторых `const`-объектов в константных выражениях отличаются; этот вопрос рассматривается в следующем разделе.

### Указатель на const

```
int main(void)
{
    int raw = 240;
    const int *view = &raw;

    int saved = *view;
    raw = 250;

    return saved != 240 || *view != 250;
}
```

`view` имеет тип «указатель на `const int`». Через него можно читать объект, но нельзя изменить его выражением `*view = ...`. Сам указатель можно перенаправить на другой подходящий объект.

Этот тип указателя не делает исходный объект `raw` неизменяемым. `raw` объявлен как `int`, поэтому прямое присваивание `raw = 250;` разрешено. При последующем чтении

через `view` получается новое значение. Ограничение относится к пути доступа через данный указатель.

Указатель на `int` можно преобразовать в указатель на `const int`: функция получает более ограниченный доступ. Обратное преобразование снимает ограничение и не должно использоваться для обхода интерфейса. Попытка изменить объект, изначально объявленный с `const`, через указатель без этого квалификатора вызывает неопределённое поведение.

## Const-указатель

В объявлении `int *const pointer` квалификатор относится к самому указателю. Его значение задаётся при инициализации и затем не может изменяться, но объект `int`, на который он указывает, доступен для изменения.

```
int main(void)
{
    int raw = 240;
    int *const pointer = &raw;

    *pointer = 250;
    return raw != 250;
}
```

Присваивание `pointer = ...` здесь недопустимо, а `*pointer = 250;` разрешено. Эти два ограничения можно совместить: `const int *const pointer = &raw;` запрещает изменение значения указателя и изменение объекта через него.

| Объявление | Можно перенаправить указатель | Можно изменить объект через указатель |
|---|---|---|
| `int *pointer` | Да | Да |
| `const int *pointer` | Да | Нет |
| `int *const pointer` | Нет | Да |
| `const int *const pointer` | Нет | Нет |

Таблица 7: Const-указатель

Таблица описывает ограничения типов; любое разрешённое обращение дополнительно требует существующего объекта и подходящего значения указателя.

## Const в интерфейсе функции

Если функция только читает массив, параметр `const int *raw` явно выражает это ограничение. Через такой параметр нельзя случайно перезаписать элемент исходного массива. Выходной параметр, наоборот, сохраняет тип `double *`, поскольку предназначен для записи результата.

```
#include <stddef.h>

int average_temperature(const int *raw, size_t count, double *output)
{
    if (raw == NULL || output == NULL || count == 0) {
```

```
        return 0;
    }

    double sum = 0.0;
    size_t valid_count = 0;

    for (size_t i = 0; i < count; ++i) {
        if (raw[i] >= 0 && raw[i] <= 500) {
            sum += raw[i] / 10.0;
            ++valid_count;
        }
    }

    if (valid_count == 0) {
        return 0;
    }

    *output = sum / valid_count;
    return 1;
}

int main(void)
{
    const int raw[] = {240, 250, 260, -1, 280};
    size_t count = sizeof raw / sizeof raw[0];
    double average = 0.0;

    if (average_temperature(raw, count, &average) == 0) {
        return 1;
    }

    return average != 25.75;
}
```

Функция читает исходные значения в десятых долях градуса, пропускает значения вне диапазона `0..500` и записывает среднее в `average`. Массив в этом примере содержит неизменяемый набор исходных данных. Тот же интерфейс принимает и обычный массив `int`: функция сможет читать его элементы через параметр, но не изменять их.

Запись `const int *raw` не фиксирует значение самого параметра-указателя и не запрещает изменение массива другими частями программы. Она описывает доступ через этот параметр. Для параметра `const int *const raw` дополнительно запрещалось бы перенаправлять локальный указатель внутри функции; на возможность вызывающей стороны передавать массив это не влияет.

## 3.3. Константные выражения и constexpr в C++

**Константное выражение** — выражение, удовлетворяющее правилам языка для вычисления значения при трансляции программы. Такие выражения нужны, например, для значений констант перечисления, меток case и размера обычного массива в C++ [3, 5].

В C11 const ограничивает изменение объекта, но само по себе не делает его имя целочисленным константным выражением. Поэтому объявление `const int count = 5;` не позволяет использовать count как значение константы enum в C. Для целочисленной константы можно применить уже изученное перечисление:

```
enum { SAMPLE_COUNT = 5 };

int main(void)
{
    int raw[SAMPLE_COUNT] = {240, 250, 260, -1, 280};
    int first = raw[0];

    return 0;
}
```

Перечисление здесь не имеет имени типа; требуется только именованная константа SAMPLE_COUNT. Её значение известно при компиляции.

В C++17 спецификатор **constexpr** явно требует, чтобы объект был инициализирован константным выражением. Объект constexpr также является const. Примеры C++ сохраняйте в main.cpp и собирайте в режиме C++17, как в 2.2.

```
int main()
{
    constexpr int sample_count = 5;
    constexpr int alarm_raw = 270;
    int raw[sample_count] = {240, 250, 260, -1, 280};
    int alarm = raw[4] >= alarm_raw;

    return 0;
}
```

sample_count задаёт размер массива, а alarm_raw — порог тревоги. Массив raw остаётся изменяемым: свойство размера массива не ограничивает изменение его элементов.

const в C++ тоже может участвовать в константных выражениях: например, подходящий целочисленный объект с константным инициализатором. Различие состоит в требовании к объявлению: const допускает инициализацию значением, полученным во время выполнения, а constexpr требует константного инициализатора.

## Функция constexpr

Спецификатор constexpr у функции позволяет вычислить её вызов в константном выражении, когда аргументы и выполняемые действия удовлетворяют соответствующим требованиям.

```
constexpr double to_celsius(int raw)
{
    return raw / 10.0;
}

int main()
{
    constexpr double alarm_threshold_c = to_celsius(270);
    int raw_temperature = 253;
    double temperature_c = to_celsius(raw_temperature);
    int alarm = temperature_c >= alarm_threshold_c;

    return 0;
}
```

Инициализатор alarm_threshold_c обязан быть константным выражением; результат равен 27.0. Ту же функцию можно вызвать с обычным значением во время выполнения. constexpr не означает, что каждый её вызов обязательно вычисляется при компиляции. Решение оптимизатора вычислить заранее обычное выражение и требование языка к константному выражению — разные вещи.

В C11 спецификатора constexpr нет. Здесь он используется как средство C++ для задания параметров и вычислений, которые должны быть доступны при компиляции.

## 3.4. `static` и `extern` в разных контекстах

Ключевые слова static и extern участвуют в объявлениях. Их смысл зависит от места объявления. Кроме области видимости и длительности хранения, здесь важно **связывание имени** (linkage): правило, определяющее, могут ли объявления в разных местах обозначать одну сущность — объект или функцию.

Имя с внешним связыванием может обозначать одну сущность в разных единицах трансляции. Имя с внутренним связыванием связывает объявления одной сущности только в пределах одной единицы трансляции. Единица трансляции — исходный файл после обработки директив препроцессора; подробнее организация файлов рассматривается в главе 4. Имя без связывания не объединяет отдельные объявления таким способом, например имя обычной локальной переменной.

### Локальная переменная `static`

Обычный локальный объект с автоматической длительностью хранения создаётся при входе в соответствующий блок и прекращает существование при выходе из него. Локальный объект, объявленный с `static`, имеет **статическую длительность хранения**: он существует в течение всего выполнения программы. Выход из функции не уничтожает его, и следующий вызов обращается к тому же объекту.

При этом область видимости имени остаётся локальной. Другие функции не получают доступа к имени только потому, что объект существует долго.

Следующая полная программа C11 накапливает число достоверных измерений. Температура задаётся в десятых долях градуса, допустимый диапазон — от `0` до `500` включительно.

```c
#include <limits.h>

int count_valid_samples(int raw)
{
    static int count;

    if (raw >= 0 && raw <= 500 && count < INT_MAX) {
        count++;
    }

    return count;
}

int main(void)
{
    int first = count_valid_samples(240);
    int second = count_valid_samples(-1);
    int third = count_valid_samples(280);

    /* first == 1; second == 1; third == 2. */
    return 0;
}
```

Объект `count` существует до, во время и после каждого вызова функции. Отсутствие явного инициализатора для объекта статической длительности хранения означает нулевую инициализацию; перед первым вызовом `count` равен `0`. Первое допустимое измерение увеличивает его до `1`, ошибочное измерение не изменяет, следующее допустимое увеличивает до `2`.

`INT_MAX` из заголовка `<limits.h>` — наибольшее значение типа `int` для данной реализации. Проверка ограничивает накопление: после достижения этого значения счётчик остаётся равным `INT_MAX`. Это исключает переполнение, но означает, что счётчик больше не отражает точное число измерений после насыщения.

В C инициализация такого объекта выполняется до начала выполнения программы, а не заново при каждом входе в функцию. Инициализатор объекта статической длительности хранения должен удовлетворять ограничениям C на постоянные значения; например, локальную переменную `static int count = 5;` объявить можно, а инициализировать её результатом вызова обычной функции нельзя. В C++17 локальный static может иметь инициализатор, вычисляемый при первом проходе через его объявление; это отличие от показанного правила C.

Замена `static int count;` на `int count = 0;` изменит поведение: счётчик будет начинать с нуля при каждом вызове. Для сохранения накопленного результата недостаточно просто использовать одинаковое имя переменной.

## `static` на уровне файла

Вне функций `static` у переменной задаёт внутреннее связывание имени. Сам объект имеет статическую длительность хранения. Обычная переменная, определённая вне функций без `static`, также имеет статическую длительность хранения, но её имя обычно имеет внешнее связывание. Следовательно, здесь `static` определяет возможность обращения по имени из других единиц трансляции, а не продлевает существование объекта.

На уровне файла `static` можно применять и к функциям: их имена получают внут-

реннее связывание.

Полная программа C11:

```c
static int alarm_threshold = 270;

static int is_alarm(int raw)
{
    return raw >= alarm_threshold && raw <= 500;
}

int main(void)
{
    int first = is_alarm(260);
    int second = is_alarm(280);

    /* first == 0; second == 1. */
    return 0;
}
```

Имена `alarm_threshold` и `is_alarm` имеют внутреннее связывание. Другой отдельно компилируемый `.c`-файл не может обратиться к этим сущностям по данным именам через объявление `extern`. В том другом файле можно определить собственную функцию или переменную с таким же именем и внутренним связыванием: это будут отдельные сущности.

Так организуют внутренние детали модуля. Связывание не является защитой памяти: оно определяет связь объявлений по имени, а не запрещает доступ к объекту через переданный указатель.

## Объявление с `extern`

**Объявление** сообщает компилятору имя и тип сущности. **Определение объекта** дополнительно задаёт сам объект, для которого требуется хранение. Объявление `extern int alarm_threshold;` обычно сообщает о существующем объекте, но не определяет его и не создаёт отдельную переменную.

Полная программа C11, в которой объявление и определение находятся в одном файле:

```c
extern int alarm_threshold;

int is_alarm(int raw)
{
    return raw >= alarm_threshold && raw <= 500;
}

int alarm_threshold = 270;

int main(void)
{
    int alarm = is_alarm(280);
    /* alarm == 1. */
    return 0;
}
```

Объявление делает имя известным в точке определения `is_alarm`, а последующее

определение задаёт объект и его начальное значение. Обе записи обозначают одну переменную. В многофайловой программе объявление можно поместить в заголовочный файл, а определение оставить в одном `.c`-файле. Такое разделение рассматривается в следующей главе.

Если программа использует объект с внешним связыванием, но ни один подключённый к сборке модуль его не определяет, обычно возникает ошибка компоновки о неопределённом символе. Совпадения имени недостаточно: объявления одной сущности должны иметь совместимые типы.

`extern` не всегда означает отсутствие определения: запись `extern int alarm_threshold = 270;` на уровне файла является определением благодаря инициализатору. Внутри блока инициализатор для объявления объекта с `extern` не допускается.

Также `extern` не обязательно меняет связывание на внешнее. Если видимое предыдущее объявление того же имени уже задаёт внутреннее связывание, последующее `extern` сохраняет это связывание. Поэтому `extern` не позволяет сделать доступной извне ранее объявленную в файле переменную `static`.

Практический выбор зависит от назначения: локальная `static` сохраняет состояние между вызовами; `static` на уровне файла ограничивает связывание имени модулем; `extern` позволяет объявить объект без повторного определения и обратиться к общему определению.

## 3.5. `volatile`: назначение и ограничения

Квалификатор `volatile` обозначает, что доступы к объекту имеют значение для выполнения программы, даже если обычный анализ исходного кода не показывает их необходимости. Значение такого объекта может изменяться способами, неизвестными компилятору; чтение или запись также могут иметь внешние последствия.

Типичный случай во встроенной программе — доступ к регистру устройства, отображённому в адресное пространство памяти. Его значение может изменяться аппаратурой, а запись может запускать действие устройства. Описание соответствующих регистров обычно предоставляется в заголовочных файлах производителя.

### Доступ к объекту и оптимизация

Для обычной переменной компилятор может исключить повторное чтение, если значение уже известно и не могло измениться в рамках анализируемого выполнения. Например, он может использовать ранее вычисленное значение вместо повторного обращения к памяти. Это допустимо, если наблюдаемое поведение программы сохраняется.

Доступы через `volatile` должны учитываться как часть наблюдаемого поведения программы. В C11 конкретные правила того, что считается доступом к такому объекту, определяет реализация. Поэтому `volatile` не означает отключение всех оптимизаций или обещание строго определённого набора машинных инструкций. В GCC отдельные завершённые выражения с такими доступами сохраняют требуемую последовательность; при этом квалификатор не устанавливает общий порядок доступов к обычным объектам и не служит барьером памяти. Документация GCC: Volatiles.

Следующая самостоятельная программа показывает объявление объекта и два последовательных чтения:

```c
#include <stdint.h>

volatile uint32_t sample = 240;

void update_sample(void)
{
    sample = 255;
}

int main(void)
{
    uint32_t first = sample;
    update_sample();
    uint32_t second = sample;

    return 0;
}
```

Первое чтение сохраняет 240, второе — 255. Здесь обновление выполняет обычная функция: асинхронных изменений нет, поэтому программа корректна и без volatile. Объекты `first` и `second` являются обычными сохранёнными копиями; последующее изменение `sample` не обновляет их автоматически.

## Совместное применение с const

`const` и `volatile` отвечают за разные свойства. `const` запрещает изменение объекта через соответствующее выражение доступа. `volatile` требует учитывать обращения к объекту при оптимизации. Поэтому сочетание `const volatile` осмысленно: программа читает значение, которое может меняться извне, но не должна записывать его через данный интерфейс.

```c
#include <stdint.h>

uint32_t read_sample(const volatile uint32_t *source)
{
    return *source;
}

int main(void)
{
    volatile uint32_t sample = 240;
    uint32_t first = read_sample(&sample);
    sample = 255;
    uint32_t second = read_sample(&sample);

    return 0;
}
```

Функция может читать объект через `source`, но не может присвоить значение выражению `*source`. Сам исходный объект здесь не объявлен `const`, поэтому `main` вправе изменять его. Аналогичный тип доступа применяется к аппаратным регистрам, предназначенным только для чтения: `const` ограничивает действия программы, а не работу аппаратуры.

### Чего `volatile` не обеспечивает

Квалификатор не гарантирует **атомарность**, то есть выполнение операции как неделимого действия относительно других участников. Например, `counter++` включает получение старого значения, вычисление нового и его сохранение. Наличие `volatile` не превращает эту последовательность в одну неделимую операцию.

Он также не обеспечивает синхронизацию потоков, защиту от гонок данных или согласованное получение нескольких связанных значений. Для этих задач применяются атомарные операции и механизмы синхронизации. `volatile` не управляет аппаратными кэшами и не заменяет требуемые процессором барьеры памяти.

Во встроенных программах `volatile` часто используется для объектов, доступных обработчикам прерываний, и при взаимодействии с аппаратурой, включая DMA — прямой доступ устройств к памяти. Однако корректность такого взаимодействия зависит также от платформы, разрядности доступов, порядка операций и правил обмена данными. Одного квалификатора для решения всех этих задач недостаточно; конкретные схемы взаимодействия рассматриваются при изучении соответствующих механизмов МК.

Для обычных локальных вычислений добавлять `volatile` не требуется. Его выбирают по требованиям к доступам, а не для исправления произвольной ошибки программы.

## 3.6. Стек, статическая память и куча

Область видимости определяет доступность имени в исходном тексте, а время жизни — период существования объекта. **Стек**, **статическая память** и **куча** описывают распространённые способы организации хранения. Эти понятия связаны с ранее рассмотренными категориями объектов, но не заменяют правила языка: стандарт C не требует, чтобы все автоматические переменные обязательно размещались в физическом стеке.

### Стек и статическое хранение

В обычной реализации **стек вызовов** хранит сведения, необходимые для возврата из функций, и часть автоматических объектов. При вызове создаётся состояние нового вызова; после возврата связанные с ним ресурсы освобождаются. Некоторые локальные значения компилятор может держать в регистрах или исключить при оптимизации.

При вложенных вызовах состояния нескольких функций существуют одновременно. Рекурсивный вызов создаёт новое состояние, поэтому глубина рекурсии влияет на необходимую память. Большие локальные массивы также могут существенно увеличивать потребление стека. Доступный объём и способ размещения стека определяются проектом и реализацией; единого размера стека для всех программ на STM32 нет.

Объекты со статической длительностью хранения существуют на протяжении всего выполнения программы. Их хранение обычно выделяется при подготовке образа и запуске программы, а не при каждом входе в функцию. Например:

```
static int raw_buffer[5];
```

Здесь создаётся массив с заранее заданной вместимостью и нулевыми начальными значениями. Если объявление находится внутри функции, имя доступно в её блоке, но массив продолжает существовать между вызовами.

В микроконтроллере изменяемые статические объекты, стек и область динамического выделения обычно размещаются в RAM. Начальные значения статических объектов могут храниться в образе во Flash и копироваться в RAM при запуске. Неизменяемые данные могут читаться непосредственно из Flash. Однако квалификатор `const` сам по себе не назначает адрес или физический вид памяти: размещение зависит от компилятора, компоновки и проекта [2].

## Динамическое выделение и void *

**Кучей** обычно называют область, из которой реализация выделяет динамическую память. В C функции `malloc` и `free` объявлены в `<stdlib.h>`. В отличие от обычной локальной переменной, выделенный объект не прекращает существование при возврате из функции, выполнившей выделение. Его время жизни заканчивается при освобождении памяти.

Тип `void *` — указатель общего назначения для обращения к объектам разных типов. Такой указатель можно преобразовать в указатель на конкретный объектный тип. Непосредственно разыменовывать `void *` для чтения значения нельзя: тип читаемого объекта не указан. В C преобразование между `void *` и указателем на объектный тип допустимо неявно.

`malloc(size)` получает число байтов и возвращает `void *`. При успехе указатель обозначает выделенную область; при неудаче возвращается нулевой указатель. Начальные значения данных в выделенной области не заданы. Поэтому данные сначала записывают и только затем читают. В следующем примере приведение типа результата `malloc` не требуется: это код C11.

## Выделение буфера температур

```c
#include <stddef.h>
#include <stdint.h>
#include <stdlib.h>

int main(void)
{
    int source[] = {240, 250, 260, 270, 280};
    size_t count = sizeof source / sizeof source[0];

    if (count == 0 || count > SIZE_MAX / sizeof(int)) {
        return 1;
    }

    int *raw = malloc(count * sizeof *raw);
    if (raw == NULL) {
        return 1;
    }

    double sum = 0.0;
    for (size_t i = 0; i < count; ++i) {
        raw[i] = source[i];
        sum += raw[i] / 10.0;
    }
    double average_c = sum / count;
    int alarm = average_c >= 27.0;
```

```
    free(raw);
    raw = NULL;
    return alarm;
}
```

`count` задаёт число элементов, а `count * sizeof *raw` — необходимое число байтов. Операнд `sizeof *raw` здесь определяет размер `int` и не разыменовывает указатель при выполнении. Проверка с `SIZE_MAX` предотвращает переполнение при умножении: иначе запрошенный размер мог бы оказаться меньше необходимого. Нулевой размер исключён отдельно. При изменении примера число копируемых элементов не должно превышать длину `source`.

После успешного выделения каждый элемент записывается до чтения. Средняя температура равна 26 °C, поэтому `alarm` равен нулю. В этой программе число элементов определяется по исходному массиву. Тот же порядок выделения применяют, когда число элементов получают во время выполнения. Для заранее известной вместимости можно использовать обычный массив.

## Освобождение и ответственность за память

`free(raw)` освобождает область, ранее выделенную `malloc`. `free(NULL)` допустим и ничего не делает. После освобождения обращаться к элементам области нельзя. Повторно освобождать ту же область или передавать `free` адрес обычной локальной переменной также нельзя: это приводит к неопределённому поведению.

Присваивание `raw = NULL` обнуляет только эту переменную. Другие указатели на освобождённую область, если они были созданы, не становятся нулевыми автоматически. Исчезновение локальной переменной-указателя также не освобождает выделенную область. Если ни один сохранённый указатель больше не позволяет освободить выделенный блок, возникает **утечка памяти**.

Для каждого выделения нужно определить, кто отвечает за освобождение. Например, если функция возвращает выделенный буфер вызывающему коду, обязанность вызвать `free` можно возложить на вызывающую сторону и явно описать это условие. Все пути выхода после успешного выделения должны учитывать освобождение либо передачу этой ответственности.

Во встроенной программе выделение может завершиться неудачей; расход памяти и время выполнения операций зависят от используемого распределителя. Если максимальное число измерений известно, буфер фиксированной вместимости обычно упрощает контроль ресурсов. Это выбор организации программы, а не запрет динамического выделения в C.

## 3.7. Контрольные вопросы

1. Чем отличаются область видимости имени, время жизни объекта и длительность его хранения?
2. Как может объект продолжать существовать, если его имя недоступно в текущем месте программы?
3. Что означает сокрытие имени во вложенном блоке? Какой объект изменяется при обращении к совпадающему имени?

4. Почему указатель на обычную локальную переменную нельзя использовать после выхода из функции?
5. Как отличается время жизни параметра от времени жизни объекта, адрес которого передан этим параметром?
6. Что именно ограничивает const? Гарантирует ли он размещение объекта во Flash?
7. Чем отличаются `const int *p`, `int *const p` и `const int *const p`? Что можно менять в каждом случае?
8. Почему указатель на const полезен в функции анализа массива? Может ли тот же исходный объект изменяться другим способом?
9. Что происходит при попытке изменить объект, определённый как const, через указатель без const?
10. Почему const в C11 не заменяет целочисленную константу enum? В чём отличие const и constexpr в C++17?
11. Должен ли каждый вызов constexpr-функции вычисляться при компиляции?
12. Как локальный static меняет сохранение состояния между вызовами? Меняется ли область видимости имени?
13. Чем static у локальной переменной отличается от static у объекта или функции на уровне файла?
14. Для чего используется extern? Чем обычное объявление extern отличается от определения переменной?
15. Почему связывание имени и время жизни объекта нужно рассматривать отдельно?
16. Как volatile влияет на доступы к объекту? Почему он не делает операцию увеличения счётчика атомарной?
17. Чем volatile отличается от средства синхронизации? Можно ли с его помощью гарантировать порядок всех обращений к памяти?
18. Как const и volatile могут применяться к одному объекту без противоречия?
19. Какие объекты обычно связаны со стеком, статической памятью и кучей? Какие из этих способов размещения гарантированы самим языком C?
20. Почему большой локальный массив и глубокая рекурсия могут исчерпать стек?
21. Чем время жизни локальной переменной-указателя с результатом malloc отличается от времени жизни выделенного блока?
22. Что необходимо проверить перед выделением памяти для массива? Почему нужно проверять и размер, и результат malloc?
23. Почему выделенную память надо инициализировать? Что означают утечка памяти, повторное освобождение и обращение после free?
24. Кто должен освобождать выделенный блок? Как это требование следует отражать при передаче указателя между функциями?
25. Когда заранее выделенный массив удобнее динамического выделения памяти в программе микроконтроллера?

# Глава 4. Многофайловая программа

## 4.1. Разделение программы на модули: интерфейс и реализация

Исходный текст программы можно разделить на несколько файлов. Такое разделение позволяет объединять связанные функции и типы, задавать границы их использования и изменять отдельные части программы.

**Модуль** в рамках этой работы — логически связанная часть программы с определённым интерфейсом и реализацией. В C отдельной языковой конструкции «модуль» нет; здесь модуль состоит из заголовочного файла `.h` и исходного файла `.c`.

**Интерфейс** описывает доступные другим частям программы типы и функции, а также условия их использования. **Реализация** содержит тела функций и внутренние детали обработки данных. Пользователь модуля вызывает функцию по её интерфейсу; ему не требуется воспроизводить её алгоритм.

Для последующих глав используется один проект:

```
examples/lab00_pipeline/
    include/
        temperature.h
    src/
        temperature.c
        host_main.c
    build/
```

`temperature.h` описывает результат анализа температуры и объявляет функцию анализа. `temperature.c` определяет эту функцию. `host_main.c` задаёт входные данные и вызывает её. Каталог `build` предназначен для результатов сборки.

Входные значения представлены целыми числами в десятых долях градуса. Допустимый диапазон — от `0` до `500` включительно. Недопустимые измерения пропускаются. Порог тревоги задаётся в тех же единицах; измерение учитывается как тревожное, если оно не меньше порога.

Результат включает количество допустимых и тревожных измерений, среднее в градусах и статус выполнения. Возврат структуры позволяет выдать все эти данные одним результатом функции.

## 4.2. Объявления и определения

**Объявление** сообщает сведения о сущности: её имя, тип и другие характеристики. **Определение** дополнительно предоставляет тело функции либо создаёт объект. Определение также является объявлением, но не всякое объявление является определением.

Прототип `TempStats temperature_analyse(const int *raw, size_t count, int threshold_raw);` объявляет функцию: сообщает тип результата и типы параметров. Тела функции в нём нет. Определение содержит тот же интерфейс и блок тела.

В исходном файле, вызывающем функцию, должно быть доступно её корректное объявление до вызова. Само определение может находиться в другом файле. Повторять прототип вручную в каждом исходном файле неудобно: при изменении интерфейса возникает риск несовпадающих объявлений. Общий заголовок позволяет использовать одну запись интерфейса во всех файлах.

Определения типов также входят в интерфейс. Если вызывающая функция создаёт объект `TempStats` и обращается к его полям, ей нужен полный состав структуры, а не только её имя.

## 4.3. Заголовочные файлы и включение через #include

Заголовок проекта `include/temperature.h` содержит:

```
#ifndef LAB00_TEMPERATURE_H
#define LAB00_TEMPERATURE_H

#include <stddef.h>

enum TempStatus {
    TEMP_OK,
    TEMP_NO_DATA,
    TEMP_BAD_ARGUMENT
};

typedef struct TempStats {
    size_t valid_count;
    size_t alarm_count;
    double average_c;
    enum TempStatus status;
} TempStats;

TempStats temperature_analyse(const int *raw, size_t count, int threshold_raw);

#endif
```

`<stddef.h>` предоставляет `size_t`. Заголовок подключает его самостоятельно, поскольку использует этот тип в собственном интерфейсе. Пользователь должен иметь возможность подключить `temperature.h`, не угадывая, какие заголовки предварительно нужны.

`enum TempStatus` определяет состояния результата. `TEMP_OK` обозначает успешный анализ; `TEMP_NO_DATA` — отсутствие допустимых измерений; `TEMP_BAD_ARGUMENT` — нарушение требований к аргументам. Значение среднего следует использовать только при `TEMP_OK`.

`typedef` вводит имя `TempStats` для типа `struct TempStats`. Поэтому пользователь может писать `TempStats stats`, без повторения слова `struct`.

Директива `#include` обрабатывается препроцессором. Содержимое найденного заголовка включается в обрабатываемый исходный текст. Заголовок не является отдельно выполняемой частью программы и обычно не собирается как самостоятельный `.c`-файл.

Запись `#include <stddef.h>` применяется для стандартного заголовка, а `#include "temperature.h"` — для заголовка проекта. Способы поиска файлов определяет реализация средств сборки. При использовании GCC параметр `-Iinclude` добавляет каталог `include` в пути поиска. Кавычки сами по себе не означают, что файл из любого каталога проекта будет найден.

## 4.4. Защита заголовков от повторного включения

Заголовок может включаться напрямую и через другие заголовки. Повторное включение его определений в один исходный текст способно вызвать ошибки, например повторное определение одной структуры.

Конструкция `#ifndef LAB00_TEMPERATURE_H` проверяет, что макрос ещё не определён. При первом включении директива `#define` определяет его, после чего обрабатывается содержимое до `#endif`. При повторном включении этого заголовка в той же единице трансляции условие ложно, и содержимое пропускается.

Такую конструкцию называют **защитой от повторного включения**, или include guard. Имя защитного макроса выбирается уникальным для заголовка: случайное совпадение с защитой другого файла может привести к пропуску его содержимого.

Защита действует внутри каждой единицы трансляции. Она не запрещает включить заголовок в два разных `.c`-файла; именно это и требуется для общего интерфейса. Она также не устраняет проблему определения одной внешней функции в нескольких единицах трансляции.

## 4.5. Реализация и связь имён между единицами трансляции

**Единица трансляции** — исходный файл вместе с текстом подключённых заголовков после препроцессирования. В проекте `temperature.c` и `host_main.c` образуют две единицы трансляции. Их обработка выполняется отдельно, а результаты затем объединяются при компоновке.

Файл `src/temperature.c` содержит реализацию:

```
#include "temperature.h"

TempStats temperature_analyse(const int *raw, size_t count, int threshold_raw)
{
    TempStats result = {0, 0, 0.0, TEMP_NO_DATA};
    double sum_c = 0.0;

    if (threshold_raw < 0 || threshold_raw > 500) {
        result.status = TEMP_BAD_ARGUMENT;
        return result;
    }
    if (count == 0) {
        return result;
    }
    if (raw == NULL) {
        result.status = TEMP_BAD_ARGUMENT;
        return result;
    }

    for (size_t i = 0; i < count; ++i) {
        if (raw[i] < 0 || raw[i] > 500) {
            continue;
        }
        ++result.valid_count;
        sum_c += raw[i] / 10.0;
        if (raw[i] >= threshold_raw) {
            ++result.alarm_count;
```

```
        }
    }

    if (result.valid_count > 0) {
        result.average_c = sum_c / result.valid_count;
        result.status = TEMP_OK;
    }
    return result;
}
```

Подключение собственного заголовка позволяет компилятору сопоставить определение с объявленным интерфейсом. Если тип результата или параметра расходится с заголовком, ошибка обнаруживается при обработке этого исходного файла.

При неверном пороге функция выдаёт `TEMP_BAD_ARGUMENT`. При допустимом пороге и `count == 0` результат имеет статус `TEMP_NO_DATA`; указатель при этом может быть нулевым. Для ненулевого количества элементов требуется ненулевой указатель на как минимум `count` доступных элементов. Проверка `NULL` не подтверждает фактическую длину массива: её обязан правильно задать вызывающий код.

После цикла отсутствие допустимых измерений сохраняет `TEMP_NO_DATA`. При наличии данных устанавливаются среднее и `TEMP_OK`.

Имя этой функции имеет **внешнее связывание**: объявления имени в разных единицах трансляции могут обозначать одну функцию. Вызов из `host_main.c` связывается с её определением в `temperature.c`. Для такой используемой функции в программе требуется одно внешнее определение.

Внутренние вспомогательные функции модуля можно определять с `static`, чтобы их имена имели внутреннее связывание и не обозначали функции других единиц трансляции. Они не включаются в общий интерфейс, если пользователю модуля не нужны.

В обычном заголовке размещают объявления внешних функций, определения типов и макросы, но не тела обычных внешних функций и не определения изменяемых глобальных объектов. Включение такого определения в несколько `.c`-файлов создало бы несколько внешних определений одной сущности. Если модуль предоставляет глобальный объект, его объявление с `extern` размещают в заголовке, а единственное определение — в реализации. В этом проекте глобальные данные для обмена не нужны: вход передаётся параметрами, выход возвращается структурой.

Исходные `.c`-файлы включать через `#include` в этом проекте нельзя. Каждый из них передаётся средствам сборки отдельно. Если включить `temperature.c` в `host_main.c` и одновременно собрать оба файла, определение функции окажется в двух единицах трансляции. Общие сведения передаются через заголовок.

## 4.6. Использование модуля и сборка проекта

Файл `src/host_main.c` использует интерфейс:

```c
#include "temperature.h"

int main(void)
{
    const int raw[] = {240, 250, 260, -1, 280};
    size_t count = sizeof raw / sizeof raw[0];
    TempStats stats = temperature_analyse(raw, count, 270);

    if (stats.status != TEMP_OK) {
        return 1;
    }
    /* Поставьте точку останова на return и посмотрите stats. */
    return 0;
}
```

После вызова `stats.valid_count` равен `4`, `stats.alarm_count` — `1`, `stats.average_c` — `25.75`, а `stats.status` — `TEMP_OK`. Значение `-1` пропущено; тревожное допустимое измерение — `280`. Для наблюдения результата в отладчике ставится точка останова на строке `return 0;`, после выполнения инициализации `stats`.

Для сборки на компьютере из каталога `examples/lab00_pipeline` сначала создаётся каталог результатов, если он ещё отсутствует:

```
New-Item -ItemType Directory -Force build
```

Затем GCC получает оба исходных файла:

```
gcc -std=c11 -Wall -Wextra -g -O0 -Iinclude src/host_main.c src/temperature.c -o
↪  build/temperature_host.exe
```

`-std=c11` задаёт стандарт языка, `-Wall -Wextra` включают предупреждения, `-g` добавляет сведения для отладки, а `-O0` отключает большинство оптимизаций. `-Iinclude` задаёт путь поиска заголовка; `-o` задаёт имя результата. Успешная программа возвращает `0`; при неожиданном статусе анализа — `1`.

Команда выполняет обработку исходных файлов и объединение результатов. Далее эти этапы будут рассмотрены отдельно. Сборка здесь предназначена для компьютера; выбор целевого микроконтроллера требует другого набора параметров и инфраструктуры проекта.

## 4.7. Контрольные вопросы

1. Чем интерфейс модуля отличается от его реализации? Что пользователь должен знать о `temperature_analyse` помимо её прототипа?
2. Чем объявление функции отличается от определения? Где они находятся в проекте?
3. Почему `temperature.h` самостоятельно подключает `<stddef.h>`?
4. Что делает `#include`? Почему заголовок не является отдельно выполняемым модулем?
5. Как работает защита `LAB00_TEMPERATURE_H`? Почему она не устраняет повторные определения внешней функции в разных исходных файлах?
6. Что входит в единицу трансляции `host_main.c` после препроцессирования?
7. Какую роль играют внешнее и внутреннее связывание имён функций?

8. Почему обычное определение внешней функции не следует помещать в общий заголовок?
9. Почему в этом проекте подключается `temperature.h`, а не `temperature.c`?
10. Что означают статусы при пустом массиве, неверном пороге и отсутствии допустимых измерений?
11. Какие аргументы нужны функции для безопасного обхода массива? Что она не может проверить по указателю?
12. Почему для сборки передаются оба `.c`-файла, а каталог `include` указывается отдельно?

# Глава 5. Компиляция, объектные файлы и компоновка

## 5.1. От исходного текста к исполняемой программе

Исходные файлы `.c` содержат описание действий на языке C. Процессор не выполняет этот текст непосредственно. **Сборка** преобразует исходные файлы и необходимые готовые компоненты в программу для выбранной платформы.

В этой главе используется проект `examples/lab00_pipeline` из предыдущей главы: интерфейс `include/temperature.h`, реализация `src/temperature.c` и основная программа `src/host_main.c`.

Далее команды выполняются в PowerShell из каталога проекта. Предполагается, что установлен GCC для компьютера и программы `gcc`, `ar` и `nm` доступны по именам. Для Windows можно использовать соответствующую сборку MinGW-w64; конкретный набор инструментов определяется выбранной средой разработки.

Сначала создают каталог для результатов:

```
New-Item -ItemType Directory -Force build
```

Сборку удобно разделять на четыре этапа:

1. Препроцессор обрабатывает директивы, включая `#include` и условную компиляцию.
2. Компилятор анализирует C-код и формирует код целевой архитектуры, который можно представить текстом на языке ассемблера.
3. Ассемблер переводит ассемблерный текст в объектный файл.
4. Компоновщик объединяет объектные файлы и библиотеки, разрешает обращения между ними и формирует конечный файл.

GCC позволяет выполнить всю последовательность одной командой или остановиться после выбранного этапа. Соответствующие опции описаны в GNU GCC: Overall Options. В обычной сборке промежуточный ассемблерный текст не обязательно сохраняется как отдельный файл. Сам термин «компилятор» часто используют для всей команды сборки, но для поиска ошибок важно различать этапы.

## 5.2. Препроцессирование

Опция `-E` останавливает обработку после препроцессора:

```
gcc -std=c11 -Iinclude -E src/temperature.c -o build/temperature.i
```

`-std=c11` выбирает стандарт языка C11. `-Iinclude` добавляет каталог `include` к путям поиска заголовков. `-o` задаёт имя выходного файла; расширение `.i` принято для препроцессированного C-текста.

В результате вместо директив `#include` в обработанном тексте присутствует содержимое подключённых заголовков. В зависимости от директив условной компиляции некоторые части текста включаются, а другие исключаются. Макросы заменяются согласно правилам препроцессора. Результат может быть значительно длиннее исходного

файла, поскольку стандартный заголовок подключает необходимые объявления и определения.

Файл `.i` ещё не является машинным кодом. Например, в нём можно найти определение `TempStats` и тело `temperature_analyse`. GCC также помещает служебные отметки, позволяющие связать обработанный текст с исходными файлами и номерами строк.

Проверка этого этапа полезна, если непонятно, какой заголовок подключён или какая ветвь условной компиляции реально попала в сборку. Если нужного файла нет в путях поиска, обработка прекращается ещё до компиляции C-кода.

## 5.3. Компиляция и ассемблирование

Опция `-S` сохраняет ассемблерное представление и останавливает сборку до ассемблирования:

```
gcc -std=c11 -g -O0 -Iinclude -S src/temperature.c -o build/temperature.s
```

Ассемблерный текст содержит инструкции целевой архитектуры и директивы, необходимые для организации кода и данных. Компилятор компьютера формирует инструкции для компьютера. Он не становится компилятором STM32 из-за того, что исходная программа предназначена для обработки данных микроконтроллера.

На этом этапе проверяются правила языка: корректность объявлений, выражений, типов и определений функций. Синтаксическая ошибка, например незакрытая скобка, препятствует получению корректного результата. Предупреждение может указывать на сомнительный код, даже если сборка продолжается. Для учебных проектов полезно дополнительно включать `-Wall -Wextra`; наличие или отсутствие предупреждений не доказывает правильность алгоритма.

Опция `-c` выполняет обработку до объектного файла включительно, но не запускает компоновку:

```
gcc -std=c11 -g -O0 -Iinclude -c src/temperature.c -o build/temperature.o
gcc -std=c11 -g -O0 -Iinclude -c src/host_main.c -o build/host_main.o
```

Каждый `.c`-файл обрабатывается независимо как отдельная единица трансляции. Заголовок обеспечивает известность интерфейса в каждой из них, но сам по себе не заменяет объектный файл с реализацией функции.

**Объектный файл** содержит скомпилированный код и данные, сведения о символах и информацию для последующего соединения с другими файлами. Он обычно ещё не готов к самостоятельному запуску. Например, `host_main.o` содержит вызов `temperature_analyse`, но её реализация расположена в `temperature.o`.

Сборка отдельных файлов позволяет после изменения одной реализации пересобрать её объектный файл и повторить компоновку. Если изменился заголовок, пересборки требуют все зависящие от него единицы трансляции. Системы сборки автоматизируют отслеживание таких зависимостей.

## 5.4. Символы и компоновка

**Символ** связывает имя сущности со сведениями, необходимыми инструментам сборки. В объектном файле символ может быть определён, например для функции с телом,

или оставаться неопределённым, если файл использует реализацию из другого места.

Символы исследуются утилитой GNU nm:

```
nm build/temperature.o
nm build/host_main.o
```

В выводе GNU `nm` часто встречаются обозначения:

| Обозначение | Типичный смысл |
|---|---|
| `T` | Определённый символ в области кода |
| `D` | Определённый символ в области инициализированных данных |
| `B` | Определённый символ в области данных, получающих нулевые начальные значения |
| `U` | Неопределённый символ: определение требуется из другого файла или библиотеки |

Таблица 8: 5.4. Символы и компоновка

Точные детали зависят от формата файла и платформы. Нижний регистр для многих категорий обозначает локальный символ. Не каждый объектный файл содержит все перечисленные категории. Здесь ожидается определение `temperature_analyse` в объектном файле реализации и обращение к нему в объектном файле программы.

Компоновка двух объектных файлов:

```
gcc build/host_main.o build/temperature.o '-Wl,-Map=build/temperature_host.map' -o
↪  build/temperature_host.exe
```

Драйвер `gcc` вызывает компоновщик и подключает стандартные компоненты выбранной среды. Опция `-Wl,` передаёт последующие параметры компоновщику. В PowerShell параметр с запятыми взят целиком в одинарные кавычки. `-Map=...` создаёт карту компоновки — текстовый отчёт о составе результата и размещении его частей.

Компоновщик сопоставляет обращения к символам с их определениями и уточняет зависимые от размещения адреса. Такие уточнения описываются **релокациями**. Он также учитывает библиотеки и правила размещения, действующие для выбранной платформы.

Запуск программы для компьютера:

```
./build/temperature_host.exe
```

Программа завершается с кодом `0`, если анализ имеет статус `TEMP_OK`; числового вывода в терминал у неё нет. Поля `stats` наблюдаются в отладчике на строке `return 0;`, как показано в главе 4. Сборка из отдельных объектов должна давать тот же результат, что и сборка одной командой.

Исполняемый файл для Windows не обязан иметь формат ELF. Формат конечного образа и его исследование рассматриваются в следующей главе на примере программы для микроконтроллера.

## 5.5. Ошибки разных этапов

Если компилятор не знает тип или обнаруживает неверный синтаксис, исправлять нужно исходный текст либо доступность необходимых объявлений. Компоновщик не может восстановить пропущенные правила языка.

Если объявление функции известно и вызов скомпилирован, но реализация не включена в сборку, ошибка возникает при компоновке. Например, следующая намеренно неполная команда не содержит `temperature.o`:

```
gcc build/host_main.o -o build/missing_implementation.exe
```

В данном проекте она должна завершиться ошибкой о неразрешённом обращении к `temperature_analyse`. Конкретные слова сообщения, оформление имени и путь к инструменту зависят от платформы и версии набора инструментов. Нужно проверить, где находится определение и включён ли содержащий его объектный файл или библиотека в команду.

Обратная проблема — несколько определений одной сущности с внешним связыванием. Например, повторное включение одного объектного файла с определением внешней функции обычно выявляется компоновщиком:

```
gcc build/host_main.o build/temperature.o build/temperature.o -o
↪  build/duplicate_implementation.exe
```

Эта команда приведена для демонстрации ошибки и не относится к нормальной сборке. Частая причина аналогичной проблемы — определение обычной внешней переменной или функции в заголовке, подключённом несколькими `.c`-файлами. Объявления интерфейса размещают в заголовке, а соответствующее определение — в реализации.

Даже успешная компоновка не гарантирует правильность типов между файлами или алгоритма. Общий заголовок нужен, чтобы компилятор проверял реализацию и её использование относительно одного интерфейса.

## 5.6. Статические библиотеки

**Статическая библиотека** — архив объектных файлов, из которого компоновщик может брать необходимые компоненты. Она позволяет передавать и использовать скомпилированные реализации группой.

Создание архива:

```
ar rcs build/libtemperature.a build/temperature.o
```

Команда GNU `ar` добавляет объектный файл в архив и создаёт индекс символов. Библиотека не запускается отдельно и не заменяет программу с `main`.

Использование библиотеки:

```
gcc build/host_main.o -Lbuild -ltemperature -o build/temperature_library_host.exe
```

`-Lbuild` добавляет каталог поиска библиотек. `-ltemperature` запрашивает библиотеку по имени; для статического архива в примере используется файл `libtemperature.a`.

Порядок аргументов важен. При обычной обработке статического архива GNU-компоновщик выбирает его объектные файлы для уже известных неразрешённых обращений. Поэтому сначала указан `host_main.o`, которому нужна функция, затем библиотека с её определением. Архив, поставленный раньше использующих его файлов, может не дать ожидаемого результата. Более сложные зависимости между библиотеками рассматриваются отдельно; для этой работы достаточно указанного порядка.

На компьютерах также применяются разделяемые библиотеки, подключаемые с участием загрузчика и операционной системы. Для рассматриваемой программы микроконтроллера такая схема не требуется.

## 5.7. Оптимизация и целевая платформа

`-O0` отключает большинство оптимизаций и удобен для начального разбора программы. `-O2` включает оптимизации, которые могут удалить лишние вычисления, объединить операции и изменить расположение кода. Компилятор должен сохранять наблюдаемое поведение корректной программы в рамках правил языка; исходные инструкции при этом не обязаны соответствовать машинным инструкциям один к одному.

Опция `-g` добавляет отладочную информацию и независима от выбора оптимизации. Сборка с `-g -O2` возможна, но отладчик может показывать, что переменная оптимизирована или выполнение не соответствует простому последовательному обходу строк. `volatile` не является общим способом отключить оптимизацию или заставить каждый шаг программы сохраняться в исходной форме.

Для STM32 нужен **кросс-компилятор**: инструмент, работающий на компьютере и создающий код для другой платформы. Для Cortex-M4 команды используют семейство `arm-none-eabi-gcc`; параметры `-mcpu=cortex-m4 -mthumb` выбирают процессор и набор инструкций. Все соединяемые компоненты должны быть совместимы по архитектуре и ABI — правилам представления данных и взаимодействия скомпилированных функций.

В учебной сборке может применяться `-mfloat-abi=soft`: операции с вещественными значениями реализуются с помощью программных средств. Компилятор может создавать обращения к вспомогательным функциям, предоставляемым его библиотекой поддержки, например `libgcc`. Даже если программист не написал такой вызов явно, нужный компонент должен попасть в компоновку. Для запуска на микроконтроллере также потребуются правила размещения памяти и стартовый код; они рассматриваются в следующих главах.

## 5.8. Контрольные вопросы

1. Какие четыре этапа выделяют в сборке C-программы и какой результат даёт каждый этап?
2. Чем различаются опции `-E`, `-S` и `-c`?
3. Для чего применяется `-Iinclude`? Создаёт ли подключение заголовка реализацию функции?
4. Почему `host_main.o` может содержать неопределённый символ `temperature_analyse`?
5. Как отличить ошибку исходного текста от отсутствия определения при компоновке?

6. Почему обычное определение внешней функции не следует размещать в заголовке, подключаемом несколькими `.c`-файлами?
7. Какие сведения можно получить из `nm` и карты компоновки?
8. Что содержит статическая библиотека и почему порядок её указания может влиять на результат?
9. Чем различаются назначения `-g`, `-O0` и `-O2`?
10. Почему скомпилированный для компьютера объектный файл нельзя просто соединить с объектными файлами Cortex-M4?
11. Откуда могут появиться вызовы вспомогательных функций, не записанные явно в исходном тексте?

# Глава 6. Устройство готовой программы

После компоновки программа представлена машинными командами, данными и служебными сведениями. Чтобы понимать расход памяти и подготовку к запуску, нужно различать файл на компьютере, образ для прошивки и объекты в памяти микроконтроллера. Исследование готового файла связывает объявления C с результатом работы компилятора и компоновщика.

В примерах используется проект `examples/lab00_pipeline`. Перед исследованием из его каталога выполните `./scripts/build_mcu.ps1`; для этого инструменты `arm-none-eabi-*` должны быть доступны в `PATH`. Состав этой сборки разбирается в разделе 7.2. Далее исследуется созданный `build/firmware.elf`.

## 6.1. ELF и содержимое готового файла

**ELF** — формат объектных и исполняемых файлов. Наличие расширения `.elf` само по себе не доказывает, что файл является готовой программой: объектные `.o` для рассматриваемой Arm-сборки также имеют формат ELF. После компоновки `firmware.elf` содержит согласованные адреса и машинный код для выбранной архитектуры.

Внутри ELF находятся заголовок, содержимое секций и таблицы, описывающие файл. Они могут включать таблицу символов, связывающую имена с адресами, и отладочные сведения, связывающие команды с исходными файлами и строками. Эти сведения полезны человеку и инструментам, хотя процессору для выполнения арифметики имена переменных не нужны.

ELF нельзя считать последовательностью байтов, которую целиком записывают во Flash. Средство загрузки выбирает данные, предназначенные для целевой памяти, и использует сведения об адресах. Заголовки ELF и отладочные таблицы обычно остаются на компьютере.

Проверить заголовок позволяет команда:

```
arm-none-eabi-readelf -h build/firmware.elf
```

Нужно обратить внимание на формат, архитектуру `Machine`, тип файла и поле `Entry point address`. Расширение не определяет внутренний формат: обычная программа Windows, созданная настольным GCC, обычно является PE-файлом `.exe`. Переименование её в `.elf` не превращает PE в ELF и не делает код совместимым с STM32. Назначение параметров `readelf` описано в документации GNU.

## 6.2. Секции: код, данные и служебная информация

**Секция** объединяет содержимое определённого назначения. В объектных файлах имеются входные секции; компоновщик собирает их в выходные секции готового файла. Следующие имена — распространённые соглашения, используемые учебным проектом, а не обязательные имена всех программ C.

| Секция | Назначение в учебном проекте |
|---|---|
| `.isr_vector` | Таблица векторов: начальное значение указателя стека, адрес обработчика сброса и другие записи |
| `.text` | Машинные команды функций |

| Секция | Назначение в учебном проекте |
|---|---|
| `.rodata` | Неизменяемые данные входных объектных файлов, например константный массив; здесь включены в выходную `.text` |
| `.data` | Изменяемые статические объекты с сохранёнными в образе начальными значениями |
| `.bss` | Область статических объектов, которым при запуске обеспечивается нулевая инициализация |
| `.symtab`, `.strtab` | Таблица символов и связанные с ней строки |
| `.debug_*` | Отладочные сведения, если они созданы при сборке |

Таблица 9: 6.2. Секции: код, данные и служебная информация

Таблицу секций можно вывести так:

```
arm-none-eabi-readelf -S build/firmware.elf
arm-none-eabi-objdump -h build/firmware.elf
```

В выводе нужно различать размер секции, её адрес в целевой памяти и смещение внутри файла. Смещение файла отвечает на вопрос «где инструмент читает данные в ELF», а адрес памяти — «где эти данные должны находиться при работе программы». Флаги секций также помогают отличить загружаемые данные от служебной информации.

У `.bss` размер в памяти может быть ненулевым при отсутствии соответствующей последовательности нулевых байтов в ELF: обычно она имеет тип `NOBITS`. Стартовый код обнуляет выделенный диапазон RAM. Такой способ экономит место в образе, но не уменьшает размер объектов в RAM.

Само объявление не всегда гарантирует отдельную запись в файле: компилятор может исключить неиспользуемый объект, заменить вычисление константой или иначе оптимизировать программу. Поэтому расположение проверяют по результатам сборки, а не только по написанному типу и квалификатору.

## 6.3. Секции и сегменты

Таблица секций удобна при компоновке и анализе. ELF также может содержать **программные заголовки**, которые описывают сегменты, в том числе загружаемые `PT_LOAD`. Один сегмент способен объединять несколько секций. Поэтому секция и сегмент — разные уровни описания файла.

```
arm-none-eabi-readelf -l build/firmware.elf
```

Команда показывает программные заголовки и, при наличии, соответствие секций сегментам. Поля размера в файле и размера в памяти могут различаться: диапазон с нулевой инициализацией не требует хранения всех нулей в файле. Конкретный загрузчик решает, какие сведения ELF использовать.

## 6.4. Адреса выполнения и загрузки: VMA и LMA

**VMA** — адрес секции при выполнении программы. **LMA** — адрес размещения её загрузочных данных. Название VMA не означает, что микроконтроллер обязательно использует виртуальную память операционной системы.

Для кода, выполняемого непосредственно из Flash, VMA и LMA обычно совпадают. Для `.data` они различаются: изменяемый объект работает в RAM, но его исходное значение должно сохраняться после отключения питания, поэтому находится в образе во Flash. Стартовый код копирует начальные байты из Flash в RAM до использования объекта.

В учебной программе `g_alarm_threshold_raw` имеет начальное значение `270`. Рабочее значение находится в `.data` в RAM, а образ содержит его инициализатор во Flash. `g_boot_counter`, первоначально равный нулю, размещается в `.bss` и получает ноль при подготовке RAM. В следующей главе эти объекты используются для наблюдения старта программы.

Для `.bss` рабочая RAM нужна, а загрузочные байты нулей в образе не нужны. Поэтому `.data` расходует и Flash для начальных данных, и RAM для рабочего объекта, тогда как `.bss` в этой организации расходует RAM. Правила задания VMA/LMA и пример копирования данных приведены в GNU ld: Output Section LMA.

## 6.5. Сценарий компоновки и карта памяти

Сценарий компоновки сообщает компоновщику, куда размещать выходные секции. В проекте используется `scripts/stm32f411.ld`.

`MEMORY` описывает регионы через начало `ORIGIN` и длину `LENGTH`. Для STM32F411CEU6 основная Flash имеет размер 512 КиБ и начинается с `0x08000000`; SRAM имеет размер 128 КиБ и начинается с `0x20000000`. Верхняя граница этого диапазона SRAM, не входящая в сам диапазон, равна `0x20020000`. Эти характеристики относятся к выбранному варианту МК, а не к любому STM32. Источники: DS10314 и RM0383.

`SECTIONS` задаёт объединение входных секций и размещение выходных. Например, `.data` назначается в RAM, а её загрузочные данные — во Flash. `ENTRY` задаёт точку входа в метаданных выходного файла. На Cortex-M аппаратный сброс использует таблицу векторов; одна лишь запись `ENTRY` не заменяет её и стартовый код. Команды сценария описаны в GNU ld: MEMORY и SECTIONS.

В этом сценарии входные `.rodata` объединяются с кодом в выходную `.text`. Поэтому в готовом ELF нет отдельной `.rodata`, а константный массив находится во Flash внутри `.text`. Название выходной секции не определяет тип каждого содержащегося в ней объекта.

Для стека выделена отдельная `.stack` размером 4096 байт у верхней границы RAM. Она имеет `NOLOAD`: область резервируется в памяти, её содержимое не записывается во Flash. Проверка `ASSERT` запрещает пересечение `.data` и `.bss` с этим резервом. Она не определяет максимальную глубину вызовов и не обнаруживает фактическое переполнение стека при выполнении программы.

В сборке этого проекта с Arm GNU Toolchain 14.2.Rel1 и параметрами раздела 7.2 получены следующие значения:

| Выходная секция | Размер, байт | VMA | LMA загрузочных байтов |
|---|---|---|---|
| .isr_vector | 408 | 0x08000000 | 0x08000000 |
| .text | 2592 | 0x08000198 | 0x08000198 |
| .data | 8 | 0x20000000 | 0x08000BB8 |
| .bss | 32 | 0x20000008 | — |
| .stack | 4096 | 0x2001F000 | — |

Таблица 10: 6.5. Сценарий компоновки и карта памяти

Загружаемый образ занимает **3008 байт Flash**. Статические данные занимают **40 байт RAM**, дополнительно зарезервировано **4096 байт под стек**. Размер `.data` включает четыре байта порога и четыре байта выравнивания. Сам ELF занимает 38436 байт на диске: в него входят также таблицы, отладочные сведения и выравнивание файла. Эти числа относятся к данной сборке; после изменений их получают заново из ELF и map-файла.

## 6.6. Символы, команды процессора и расход памяти

```
arm-none-eabi-nm -n build/firmware.elf
arm-none-eabi-size build/firmware.elf
arm-none-eabi-objdump -d build/firmware.elf
arm-none-eabi-objdump -d -S build/firmware.elf
arm-none-eabi-objdump -g build/firmware.elf
```

`nm -n` сортирует символы по адресам. Так можно найти функцию `temperature_analyse` и глобальные объекты. Символ способен обозначать функцию, объект или границу секции, заданную сценарием; наличие символа не обязательно означает отдельную переменную C. Буквенные обозначения категорий описаны в GNU nm.

`size` даёт сводку размеров. Стандартные колонки `text`, `data`, `bss` группируют содержимое по правилам выбранного формата вывода, поэтому `text` не следует автоматически отождествлять исключительно с секцией `.text`. Сумма колонок не равна ни размеру ELF на диске, ни полному пиковому расходу RAM со стеком. Подробнее: GNU size.

`objdump -d` дизассемблирует код. `-S` добавляет исходный текст, если доступны отладочные сведения и исходники; `-g` выводит отладочную информацию. Исходная инструкция C может соответствовать нескольким машинным командам или исчезнуть после оптимизации. Назначение параметров приведено в GNU objdump.

## 6.7. Map-файл и исключение неиспользуемых секций

**Map-файл** — текстовый отчёт компоновщика: размещение выходных секций, вклад входных файлов и значения символов. Он не является исполняемой программой или файлом прошивки. Его создание включают на этапе компоновки через аргумент GCC:

```
'-Wl,-Map=build/firmware.map'
```

Это один аргумент команды сборки, а не самостоятельная команда PowerShell. Кавычки сохраняют его целиком. Полный вызов приведён в разделе 7.2 и в `scripts/build_`

`mcu.ps1` проекта.

Параметр `--gc-sections` удаляет недостижимые входные секции. При раздельных секциях функций и данных удаление может быть достаточно точным. Компоновщик начинает с корневых секций и следует ссылкам, описанным релокациями; сценарий может сохранить секцию через `KEEP`. Поэтому таблицу векторов, используемую аппаратно, необходимо сохранить соответствующим способом. Удаляется не произвольный «неиспользуемый файл `.o`», а секции по правилам достижимости. Map и сборка позволяют проверить результат. Источник: GNU ld: Options.

## 6.8. BIN и HEX

```
arm-none-eabi-objcopy -O binary build/firmware.elf build/firmware.bin
arm-none-eabi-objcopy -O ihex build/firmware.elf build/firmware.hex
```

**BIN** содержит сырые байты выбранного загрузочного содержимого. В нём нет таблицы символов и самостоятельного описания базового адреса: адрес прошивки задают отдельно. Расстояния между загрузочными областями могут потребовать заполнения промежутков, поэтому размер BIN нельзя получать сложением размеров всех секций ELF.

**Intel HEX** представляет данные текстовыми записями с адресами и контрольными суммами. Он удобен для передачи раздельных адресных областей; возможные записи начального адреса не заменяют требования аппаратного запуска Cortex-M. Ни BIN, ни HEX не предназначены для сохранения исходной отладочной информации ELF.

Преобразование формата не перекомпилирует программу для другого процессора и не меняет необходимые стартовые действия. Для отладки сохраняют ELF, а формат для записи выбирают по возможностям средства прошивки. Правила преобразования описаны в GNU objcopy.

## 6.9. Контрольные вопросы

1. Чем готовый ELF отличается от последовательности байтов для записи во Flash?
2. Какие данные обычно находятся в `.text`, `.rodata`, `.data` и `.bss`?
3. Почему ненулевой размер `.bss` не требует такой же последовательности нулей в образе?
4. Чем отличаются секция и сегмент `PT_LOAD`?
5. Почему для `.data` различаются VMA и LMA? Кто переносит её начальные данные?
6. Для чего служат `MEMORY`, `SECTIONS` и `ENTRY`? Заменяет ли `ENTRY` таблицу векторов?
7. Что можно узнать через `nm`, `size` и `objdump`?
8. Почему размер ELF на диске не показывает полный расход Flash и RAM?
9. Что сообщает map-файл и почему его нельзя прошивать как программу?
10. По какому принципу `--gc-sections` удаляет секции и зачем используется `KEEP`?
11. Какие сведения содержит HEX и какие сведения отсутствуют в BIN?
12. Чем различаются смещение данных внутри ELF, их загрузочный адрес во Flash и рабочий адрес в RAM?

# Глава 7. Запуск и выполнение программы

В этой главе рассматривается пример запуска модуля обработки температур на STM32F411CEU6. Входные данные заданы массивом, результат сохраняется в памяти и может наблюдаться в отладчике. Отдельная функция обработки остаётся той же, что использовалась на компьютере. Пример поясняет устройство запуска; выполнять его на плате в рамках ЛР0 не требуется.

## 7.1. Состав программы микроконтроллера

Для выполнения С-программы на микроконтроллере нужны машинный код, размещение в памяти и подготовка среды выполнения. В учебном проекте этим задачам соответствуют следующие файлы:

```
examples/lab00_pipeline/
    include/temperature.h
    src/temperature.c
    src/host_main.c
    src/main_mcu.c
    src/system_minimal.c
    startup/startup_stm32f411xe.s
    scripts/stm32f411.ld
    scripts/build_mcu.ps1
    build/
```

`host_main.c` и `main_mcu.c` — два варианта основной программы для разных платформ. В одну сборку включают только один из них. `temperature.c` используется в обеих сборках.

`startup_stm32f411xe.s` содержит таблицу векторов и обработчик сброса. В проект включён стартовый файл STMicroelectronics с сохранённой лицензией. `stm32f411.ld` задаёт размещение программы и границы областей памяти. `system_minimal.c` предоставляет функции, вызываемые при старте.

Файл `src/main_mcu.c` содержит:

```c
#include <stdint.h>
#include "temperature.h"

int g_alarm_threshold_raw = 270;
volatile uint32_t g_boot_counter;
volatile uint32_t g_done;
volatile TempStats g_statistics;

static const int raw_samples[] = {240, 250, 260, -1, 280};

int main(void)
{
    ++g_boot_counter;
    g_statistics = temperature_analyse(raw_samples,
        sizeof raw_samples / sizeof raw_samples[0], g_alarm_threshold_raw);
    g_done = 1;

    for (;;) {
        /* Результаты доступны в отладчике после g_done == 1. */
    }
}
```

Порог `g_alarm_threshold_raw` позволяет проследить инициализацию `.data`. Счётчик, признак готовности и структура результата позволяют проследить обнуление `.bss` и последующие записи из `main`. `volatile` сохраняет обращения к наблюдаемым объектам по правилам языка. Префикс `g_` — выбранное соглашение об именах глобальных объектов, а не конструкция C.

После вычисления программа остаётся в бесконечном цикле. Результат доступен до сброса или изменения памяти. `g_boot_counter` равен единице после выполнения этой программы: каждый полноценный запуск снова обнуляет `.bss`, поэтому объект не накапливает число включений питания.

## 7.2. Сборка для Cortex-M4

Нужен установленный Arm GNU Toolchain с инструментами `arm-none-eabi-*`. Команды выполняются в PowerShell из каталога `examples/lab00_pipeline`. Для приведённого проекта проверена версия **Arm GNU Toolchain 14.2.Rel1**, GCC 14.2.1.

Сначала собираются отдельные исходные файлы:

```
New-Item -ItemType Directory -Force build/mcu

$compileFlags = @('-mcpu=cortex-m4', '-mthumb', '-mfloat-abi=soft',
    '-std=c11', '-Wall', '-Wextra', '-g3', '-O0',
    '-ffunction-sections', '-fdata-sections', '-Iinclude')

arm-none-eabi-gcc @compileFlags -c src/temperature.c -o build/mcu/temperature.o
arm-none-eabi-gcc @compileFlags -c src/main_mcu.c -o build/mcu/main_mcu.o
arm-none-eabi-gcc @compileFlags -c src/system_minimal.c -o build/mcu/system_minimal.o
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -mfloat-abi=soft -g3 -c
↪  startup/startup_stm32f411xe.s -o build/mcu/startup.o
```

`-mcpu=cortex-m4 -mthumb` выбирают целевые инструкции. `-mfloat-abi=soft` задаёт программную реализацию вещественной арифметики и соответствующие правила передачи аргументов. В этой сборке не требуется включать аппаратный FPU.

`-ffunction-sections -fdata-sections` разделяют функции и данные по входным секциям, чтобы компоновщик мог удалять недостижимые части.

Затем объектные файлы соединяются:

```
arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -mfloat-abi=soft -nostartfiles
↪  --specs=nosys.specs -T scripts/stm32f411.ld '-Wl,-Map=build/firmware.map'
↪  '-Wl,--gc-sections' build/mcu/startup.o build/mcu/system_minimal.o
↪  build/mcu/main_mcu.o build/mcu/temperature.o -o build/firmware.elf
```

`-T` выбирает сценарий компоновки. `-nostartfiles` исключает стандартные стартовые объектные файлы набора инструментов: в команду уже передан собственный `startup.o`. Библиотеки поддержки компилятора и C при этом доступны. `nosys.specs` подключает заглушки системных вызовов; они не обеспечивают вывод в терминал или работу операционной системы на плате.

Для повторной сборки можно выполнить готовый сценарий:

```
./scripts/build_mcu.ps1
```

Он проверяет успешность каждого шага, создаёт ELF, map-файл, BIN и HEX. Если инструменты не добавлены в `PATH`, их каталог можно передать через `-ToolchainBin`.

Объекты микроконтроллера находятся в `build/mcu`, отдельно от объектов компьютера из главы 5. Это позволяет возвращаться к обеим сборкам без подмены объектного файла кодом другой архитектуры.

## 7.3. Сброс и таблица векторов

**Сброс** приводит микроконтроллер к начальному состоянию, после которого начинается выполнение стартового кода. Для рассматриваемого запуска выбирается основная Flash: у STM32F411 этому соответствует `BOOT0 = 0`. Конкретное положение перемычки или способ задания уровня определяются схемой платы. Режимы загрузки описаны в RM0383, раздел Boot configuration.

После сброса ядро Cortex-M получает из первых двух слов таблицы векторов начальное значение **MSP** — основного указателя стека — и адрес обработчика сброса. При загрузке из основной Flash её начало отображается также по адресу `0x00000000`: чтения по `0x00000000` и `0x00000004` соответствуют словам по `0x08000000` и `0x08000004`.

В проверенной сборке первые слова имеют значения:

| Адрес во Flash | Содержимое | Назначение |
|---|---|---|
| `0x08000000` | `0x20020000` | Начальное значение MSP, верхняя граница SRAM |
| `0x08000004` | `0x08000199` | Адрес `Reset_Handler` с установленным младшим битом |

Таблица 11: 7.3. Сброс и таблица векторов

Младший бит адреса обработчика обозначает Thumb-состояние. Сами инструкции обработчика начинаются по чётному адресу `0x08000198`. Это правило адреса функции

не относится к первому слову с указателем стека. Другие записи таблицы задают адреса обработчиков исключений и прерываний. Устройство сброса и таблицы описано в PM0214.

Поле точки входа ELF в данном проекте тоже указывает на `Reset_Handler`, но аппаратный сброс получает адрес из таблицы векторов. Процессор не читает заголовок ELF, оставшийся на компьютере.

Таблица расположена в начале основной Flash, а `KEEP(*(.isr_vector))` сохраняет её при удалении недостижимых секций. Сценарий компоновки проверяет её положение и размер. Эти условия вместе со стартовым кодом связывают скомпилированную программу с аппаратным запуском.

## 7.4. Подготовка памяти и переход в main

Для выбранного стартового файла последовательность следующая:

```
аппаратный сброс → получение MSP и адреса Reset_Handler
Reset_Handler → SystemInit
              → копирование начальных данных .data из Flash в RAM
              → обнуление .bss
              → __libc_init_array
              → main
```

Компоновщик предоставляет стартовому коду символы границ:

| Символ | Значение для стартового кода |
|---|---|
| `_estack` | Начальное положение указателя стека |
| `_sidata` | Адрес начальных данных `.data` во Flash |
| `_sdata`, `_edata` | Начало и конец рабочей `.data` в RAM |
| `_sbss`, `_ebss` | Начало и конец обнуляемой `.bss` в RAM |

Таблица 12: 7.4. Подготовка памяти и переход в main

Конечные границы не входят в соответствующие диапазоны. Это символы сценария компоновки; выделять для них отдельные переменные C не требуется.

Копирование `.data` создаёт в RAM значение `g_alarm_threshold_raw = 270`. Обнуление `.bss` задаёт нулевые начальные значения счётчика, признака готовности и полей результата. Тем самым стартовый код обеспечивает условия, на которые опирается C-программа со статическими объектами.

В этом стартовом файле `SystemInit` вызывается **до** подготовки `.data` и `.bss`. Поэтому в ней нельзя полагаться на уже установленные начальные значения глобальных C-объектов. Порядок следует проверять по конкретному стартовому файлу.

Учебная программа работает после сброса из основной Flash с исходными настройками тактирования и программной вещественной арифметикой. Для неё `src/system_minimal.c` содержит две пустые функции:

```
void SystemInit(void)
{
}

void _init(void)
{
}
```

`SystemInit` — точка ранней системной настройки. В проектах с другой организацией запуска здесь может потребоваться подготовка тактирования, FPU или положения таблицы векторов. `_init` — функция, вызываемая используемой библиотечной процедурой инициализации; в этом проекте дополнительных действий у неё нет.

`__libc_init_array` выполняется после подготовки RAM и обрабатывает функции инициализации, включённые в образ. При использовании C++ сюда может относиться динамическая инициализация объектов со статической длительностью хранения, включая вызовы конструкторов. Наличие этой процедуры связывает стартовый код с библиотекой среды выполнения.

После подготовки вызывается `main`. В учебной программе она сохраняет результат и остаётся в цикле. Возврат в операционную систему здесь не предусмотрен.

## 7.5. Исходный код и машинные инструкции

При выполнении программы процессор обращается к регистрам, читает и записывает память, выполняет арифметические команды и переходы. Выражение C описывает требуемое действие, а конкретную последовательность инструкций выбирает компилятор.

В проверенной сборке фрагмент для `++g_boot_counter` выглядит так:

```
ldr  r3, [pc, #52]    ; получить адрес g_boot_counter из таблицы рядом с кодом
ldr  r3, [r3, #0]     ; прочитать текущее значение
adds r3, #1          ; прибавить единицу
ldr  r2, [pc, #44]    ; получить адрес для записи
str  r3, [r2, #0]     ; записать новое значение
```

`ldr` загружает значение, `str` сохраняет его, `adds` выполняет сложение. Здесь `r2` и `r3` — регистры процессора, `pc` используется для получения адресов относительно положения кода. Комментарии добавлены для объяснения. Полный фрагмент с адресами и байтами получается командой:

```
arm-none-eabi-objdump --disassemble=main build/firmware.elf
```

В этой же функции виден `bl ... <temperature_analyse>` — вызов анализа. Перед вызовом формируются аргументы и место для результата, после него структура копируется в `g_statistics`. Правила такого взаимодействия задаёт ABI; структура не обязана возвращаться одним регистром.

Размеры машинных команд и их число отличаются от числа инструкций C. Вещественное деление в выбранной сборке вызывает библиотечную реализацию, а бесконечный цикл превращается в повторяющийся переход. Для просмотра связи с исходными строками используется `objdump -d -S`; при оптимизации эта связь может стать менее прямой.

## 7.6. Запись программы и наблюдение результата

**Прошивка** записывает загрузочные данные в память микроконтроллера. **Отладка** позволяет останавливать выполнение, проходить код по шагам и наблюдать значения. Для аппаратного выполнения рассматриваемого примера в последующих работах предполагаются Black Pill на STM32F411CEU6 и ST-Link V2. Подключение, питание и положение `BOOT0` определяются документацией конкретной платы. Ниже приведена последовательность для OpenOCD и GDB как пример использования инструментов; студент может выбрать другие средства прошивки и отладки.

В качестве воспроизводимого варианта используются **OpenOCD 0.12.0** и GDB из Arm GNU Toolchain. В первом терминале из каталога проекта запускается сервер:

```
openocd -f interface/stlink.cfg -f target/stm32f4x.cfg
```

Во втором терминале открывается ELF:

```
arm-none-eabi-gdb build/firmware.elf
```

Далее команды вводятся в GDB:

```
target extended-remote localhost:3333
monitor reset halt
load
compare-sections
monitor reset halt
break main
continue
```

`load` записывает образ по его загрузочным адресам; `compare-sections` сравнивает записанное содержимое с образом. Повторный сброс запускает стартовую последовательность. Команды подключения и загрузки описаны в руководстве OpenOCD.

На остановке в начале `main`, до `++g_boot_counter`, нужно проверить:

```
print g_alarm_threshold_raw
print g_boot_counter
print g_done
```

Ожидаются `270`, `0`, `0`. Затем выполняйте `next` до завершения присваивания `g_done = 1`, ориентируясь по текущей строке и значению переменной. `next` проходит вызов функции целиком; `step` позволяет войти в неё. После выполнения присваивания проверяются:

```
print g_boot_counter
print g_done
print g_statistics
```

| Объект или поле | Ожидаемое значение |
|---|---|
| `g_boot_counter` | `1` |
| `g_done` | `1` |
| `g_statistics.valid_count` | `4` |
| `g_statistics.alarm_count` | `1` |
| `g_statistics.average_c` | `25.75` |

| Объект или поле | Ожидаемое значение |
|---|---|
| `g_statistics.status` | `TEMP_OK` |

Таблица 13: 7.6. Запись программы и наблюдение результата

Остановка перед инструкцией показывает состояние **до** её выполнения. Строка, выделенная в отладчике, обычно обозначает следующий шаг, поэтому результат присваивания проверяют после шага. После входа в бесконечный цикл дальнейший `next` может ожидать завершения цикла; выполнение можно остановить через `Ctrl+C`.

В графической среде используются те же действия: загрузить актуальный ELF, сбросить МК, остановиться в `main`, выполнить шаги и открыть наблюдение за переменными. Для отладки нужен именно ELF текущей сборки: старые адреса и сведения о строках могут не соответствовать записанному коду.

Команды подключения зависят от отладчика и установленного набора инструментов. Сборка и содержимое ELF проверены; последовательность работы с отладчиком должна быть дополнительно пройдена на выбранной плате курса.

## 7.7. Что проверять при переносе и ошибках запуска

Модуль `temperature.c` использует возможности C и не обращается к периферии. Его исходный текст можно собрать для другого совместимого окружения. Готовый образ требует совместимости инструкций, ABI, карты памяти, таблицы векторов и стартовой настройки. Для другого микроконтроллера подбирают стартовый файл и сценарий компоновки, затем выполняют новую сборку.

Для поиска причины сбоя полезно определить последний достигнутый этап:

| Наблюдение | Что проверить |
|---|---|
| Компоновка не находит `SystemInit`, `_init` или символы границ памяти | Состав объектных файлов, библиотек и выбранный сценарий |
| Запись завершилась, но `main` не достигается | Режим загрузки, первые слова векторов, выполнение `Reset_Handler` |
| В начале `main` порог не равен `270` или счётчик не равен нулю | Копирование `.data`, обнуление `.bss`, проведение полного сброса |
| `g_done == 1`, но результат отличается от ожидаемого | Массив, порог, алгоритм и соответствие ELF записанному образу |
| Отладчик показывает недоступную локальную переменную | Её время жизни, текущую функцию, наличие отладочных сведений и оптимизацию |

Таблица 14: 7.7. Что проверять при переносе и ошибках запуска

Точка останова в `Reset_Handler` позволяет отделить ошибку аппаратного старта от ошибки последующих действий. Точка в `main` показывает, что стартовый путь пройден. Проверка полей результата после вызова связывает запуск микроконтроллера с ранее проверенным алгоритмом C.

## 7.8. Контрольные вопросы

1. Какие файлы нужны для запуска учебной программы на STM32 и какую задачу решает каждый из них?
2. Почему `host_main.c` и `main_mcu.c` не включают в одну сборку?
3. Что задают `-mcpu`, `-mthumb`, `-mfloat-abi=soft`, `-T` и `-nostartfiles`?
4. Откуда ядро получает начальные MSP и адрес обработчика сброса? Как это связано с двумя первыми словами образа?
5. Почему адрес обработчика в таблице нечётный, а адрес его первой инструкции чётный?
6. Почему поле точки входа ELF само по себе не обеспечивает аппаратный запуск?
7. Кто создаёт значение `270` в рабочем объекте `.data` и нули в `.bss`?
8. Что обозначают `_sidata`, `_sdata`, `_edata`, `_sbss` и `_ebss`? Являются ли они отдельными C-объектами?
9. Почему выбранная `SystemInit` не должна рассчитывать на инициализированные глобальные объекты?
10. Для чего вызывается `__libc_init_array` и как это связано с C++?
11. Как одно выражение C может превратиться в несколько машинных команд или библиотечный вызов?
12. Чем прошивка отличается от отладки? Для чего отладчику нужен актуальный ELF?
13. Что нужно проверить до выполнения первой инструкции `main` и после установки `g_done`?
14. Почему `g_boot_counter` не хранит число аппаратных сбросов между запусками?
15. Какие части проекта требуют проверки и замены при переносе на другой микроконтроллер?

# Дополнительный раздел. ООП и структуры данных в C++

**Объектно-ориентированное программирование (ООП)** организует программу вокруг объектов, их состояния и доступных операций. В C++ для этого используются классы. Класс может хранить порог температуры, управлять динамическим буфером или предоставлять операции стека и очереди. Раздел рассматривает устройство собственных классов и применение стандартных средств хранения данных.

Примеры этого раздела написаны на **C++17** и выполняются на компьютере. Полную программу сохраняйте в файл `.cpp` и собирайте через `g++`:

```
g++ -std=c++17 -Wall -Wextra -g -O0 main.cpp -o main.exe
```

`g++` выбирает C++ и подключает его стандартную библиотеку при компоновке; это поведение описано в документации GCC. Определения классов размещаются перед `main`, а фрагменты их использования — внутри `main`.

## 8.1. Классы, объекты, методы и пространства имён

**Класс** определяет тип. **Объект класса** — конкретный экземпляр этого типа. Класс может содержать данные и функции-члены; последние обычно называют **методами**.

Начнём с температуры, заданной в десятых долях градуса:

```
class Temperature {
public:
    int raw = 0;

    double celsius() const
    {
        return raw / 10.0;
    }
};

int main()
{
    Temperature first;
    Temperature second;

    first.raw = 250;
    second.raw = 280;

    double first_c = first.celsius();
    double second_c = second.celsius();

    return first_c < second_c ? 0 : 1;
}
```

`Temperature` — имя типа, `first` и `second` — два разных объекта. Запись `int raw = 0;` задаёт начальное значение поля каждого нового объекта. У каждого своё поле `raw`. Изменение `first.raw` не изменяет `second.raw`.

Вызов `first.celsius()` выполняет метод для объекта `first` и возвращает `25.0`; `second.celsius()` возвращает `28.0`. Нестатический метод имеет доступ к объекту вы-

зова через указатель `this`. Здесь обращение к `raw` внутри метода означает обращение к полю этого объекта.

`const` после списка параметров позволяет вызвать метод для константного объекта. Через `this` такой метод получает доступ к объекту как к `const`; в примере он только читает поле. После закрывающей скобки определения класса ставится `;`.

Состояние объекта определяется его данными. Метод задаёт операцию над этим состоянием. Код метода не создаётся заново для каждого объекта: отдельные значения имеют нестатические поля, а определения функций общие для класса. Основные правила классов приведены в N4659, раздел class.

## Пространства имён

**Пространство имён** (`namespace`) объединяет имена типов, функций и других сущностей. Оно помогает различать одинаковые короткие имена в разных частях проекта. Например, класс `Temperature` может существовать и в модуле датчика, и в модуле моделирования:

```
#include <iostream>

namespace sensors {
    class Temperature {
    public:
        int raw = 0;

        double celsius() const
        {
            return raw / 10.0;
        }
    };
}

namespace simulation {
    class Temperature {
    public:
        double value_c = 0.0;
    };
}

int main()
{
    sensors::Temperature measured;
    measured.raw = 250;

    simulation::Temperature calculated;
    calculated.value_c = 26.5;

    std::cout << measured.celsius() << ' '
              << calculated.value_c << '\n';
    return 0;
}
```

Вывод: `25 26.5`. `sensors::Temperature` и `simulation::Temperature` — разные типы. Оператор `::` уточняет, в какой области искать имя. Для пространства имён запись `sensors::Temperature` означает тип `Temperature` из `sensors`; точка в `measured.celsius()` обращается к члену конкретного объекта.

Определение `namespace` размещается вне функций — в глобальной области или внутри другого пространства имён. Его можно продолжить другим блоком `namespace sensors { ... }`, в том числе в другом файле. Это позволяет объединить объявления и определения одного модуля. Пространство имён не создаёт объекта и не задаёт доступ `private` или `public`. C++17: определения пространств имён.

`std` — пространство имён стандартной библиотеки C++. Поэтому `std::cout`, `std::vector` и `std::size_t` имеют общий префикс. Подключение заголовка предоставляет объявления нужных сущностей, а запись `std::` уточняет их имена.

**Объявление using** позволяет использовать выбранное имя без префикса в соответствующей области. Например, внутри `main` после `using sensors::Temperature;` можно написать `Temperature sample;`: это тот же тип `sensors::Temperature`, а не новый тип. Аналогично `using std::cout;` позволяет обращаться к `cout`. C++17: using-declaration.

**Директива using namespace** влияет на поиск всех имён пространства: после `using namespace std;` можно использовать короткие имена `cout` и `vector`. Она не копирует содержимое пространства. Если поиск находит несколько подходящих одноимённых сущностей, обращение может стать неоднозначным. Поэтому в примерах методички используется явный префикс `std::`; в заголовках проекта также сохраняйте явные имена, чтобы не менять поиск имён в каждом подключающем файле. C++17: using-directive.

## 8.2. Инкапсуляция и доступ к членам класса

В первом варианте вызывающий код может записать в `raw` любое целое число. Для объекта, представляющего допустимую температуру из диапазона `0..500`, введём проверяемую операцию изменения.

**Инкапсуляция** объединяет состояние и операции над ним в одном типе и задаёт границу доступа к реализации. В этой задаче значение хранится в закрытом поле, а изменение выполняется через метод с проверкой:

В C++ `bool` — встроенный логический тип, а `false` и `true` — значения этого типа.

```
class Temperature {
public:
    bool set_raw(int raw)
    {
        if (raw < 0 || raw > 500) {
            return false;
        }
        raw_ = raw;
        return true;
    }

    int raw() const
    {
        return raw_;
    }

    double celsius() const
    {
        return raw_ / 10.0;
    }

private:
    int raw_ = 0;
};
```

Это новая версия `Temperature`, заменяющая предыдущую. Поле `raw_` доступно методам класса; внешний код использует `set_raw`, `raw` и `celsius`. Суффикс `_` в имени поля — выбранное соглашение об именовании.

```
Temperature sample;
bool accepted = sample.set_raw(250);
bool rejected = sample.set_raw(-1);
double value_c = sample.celsius();

Temperature copy = sample;
copy.set_raw(280);

const Temperature &view = sample;
double observed_c = view.celsius();
```

После этих действий `accepted == true`, `rejected == false`, `value_c == 25.0`. При отказе `set_raw` сохраняет прежнее значение. `copy` — отдельный объект со значением 280; у `sample` остаётся 250. В данном классе копирование повторяет значение поля `raw_` .

`view` обозначает исходный объект через ссылку на `const`. Через неё можно вызвать `celsius()`, но вызов `view.set_raw(300)` недопустим: этот метод допускает изменение объекта. Прямое обращение из `main` к `sample.raw_` также вызовет ошибку компиляции.

| Спецификатор | Доступ |
|---|---|
| `public` | Член входит в доступный вызывающему коду интерфейс |
| `private` | Доступ имеют члены самого класса и явно объявленные друзья класса |

| Спецификатор | Доступ |
|---|---|
| `protected` | Доступ дополнительно предоставляется производным классам и их друзьям по правилам C++ |

Таблица 15: 8.2. Инкапсуляция и доступ к членам класса

Спецификатор действует до следующего спецификатора или конца класса. Для `class` члены по умолчанию `private`, для `struct` — `public`. В C++ обе конструкции определяют классовые типы и допускают методы; здесь `struct` используется для открытых записей данных, а `class` — для типов с ограниченным интерфейсом. Правила доступа: N4659, class.access.

**Инвариант класса** — условие, которое поддерживается для допустимых состояний его объектов. У второй версии `Temperature` это `0 <= raw_ && raw_ <= 500`: начальное значение равно нулю, а метод изменения отклоняет недопустимый ввод. Закрытое поле позволяет сосредоточить эту проверку в одном месте. Ответственность за корректность операций остаётся в реализации класса.

**Геттер** предоставляет чтение свойства объекта, **сеттер** — его изменение. В примере `raw()` и `celsius()` являются геттерами, `set_raw()` — сеттером. Названия методов определяются интерфейсом: в заданиях используются также `get_raw()` и `get_celsius()`. Возвращение значения закрытого поля по значению позволяет читать его без доступа к изменению; выдача изменяемой ссылки на это поле позволила бы обходить проверку сеттера.

Один сеттер может изменять несколько связанных полей. Для диапазона с инвариантом `lower_` $\leq$ `upper_` метод `set_lower(value)` проверяет новую нижнюю границу относительно текущей верхней, а `set_upper(value)` — новую верхнюю относительно текущей нижней. Метод `set_bounds(lower, upper)` сначала проверяет всю пару, затем изменяет оба поля. При недопустимой паре сохраняются обе прежние границы. Последовательный вызов двух отдельных сеттеров не заменяет такую операцию: промежуточное состояние может нарушить инвариант.

## 8.3. Конструкторы, деструкторы и время жизни

### Конструктор и начальное состояние

**Конструктор** — специальная функция-член, которая участвует в инициализации объекта. Его имя совпадает с именем класса; тип возвращаемого значения не указывается.

Определим правило тревоги с порогом, задаваемым при создании:

```
class ThresholdAlarm {
public:
    explicit ThresholdAlarm(int threshold_raw)
        : threshold_raw_(threshold_raw)
    {
    }

    bool is_alarm(int raw) const
    {
        return raw >= threshold_raw_;
    }

private:
    int threshold_raw_;
};
```

```
ThresholdAlarm warm(270);
ThresholdAlarm hot{300};

bool first_alarm = warm.is_alarm(280);
bool second_alarm = hot.is_alarm(280);
```

`warm` хранит порог 270, `hot` — 300. Для измерения 280 первое правило даёт `true`, второе — `false`. Порог и измерение выражены в десятых долях градуса. Допустимость измерения проверяет вызывающий код; правило определяет условие тревоги сравнением целых значений.

Запись `: threshold_raw_(threshold_raw)` — **список инициализации членов**. Поле инициализируется до выполнения тела конструктора. Тело здесь пустое, потому что требуемое начальное состояние уже задано.

Если полей несколько, они инициализируются в порядке объявления в классе, независимо от порядка записей в списке. Поля-ссылки и поля `const` должны получить значение при инициализации: заменить её последующим присваиванием в теле конструктора нельзя. Правила описаны в N4659, class.base.init.

`explicit` запрещает использовать этот конструктор для неявного превращения целого числа в объект при обычной передаче аргумента или копирующей инициализации. Объект создаётся явной записью, как в примере. Поскольку у класса объявлен конструктор с обязательным параметром, запись `ThresholdAlarm rule;` не подходит: конструктора без аргументов здесь нет. Определение конструкторов: N4659, class.ctor.

## Деструктор и завершение жизни объекта

**Деструктор** выполняет действия при уничтожении объекта класса. Его имя начинается с `~`, за которым следует имя класса; у него нет параметров и типа возвращаемого значения.

Проследим порядок на полной программе:

```cpp
#include <cstdio>

class ScopeTrace {
public:
    explicit ScopeTrace(int id) : id_(id)
    {
        std::printf("create %d\n", id_);
    }

    ~ScopeTrace()
    {
        std::printf("destroy %d\n", id_);
    }

private:
    int id_;
};

int main()
{
    ScopeTrace outer(1);
    {
        ScopeTrace inner(2);
    }
    return 0;
}
```

`std::printf` из `<cstdio>` выводит строку в терминал; `%d` заменяется значением целого аргумента, `\n` задаёт перевод строки. Префикс `std::` обозначает имя из пространства имён стандартной библиотеки C++. Программа выводит:

```
create 1
create 2
destroy 2
destroy 1
```

У `inner` автоматическая длительность хранения: выход из внутреннего блока вызывает его деструктор. `outer` уничтожается при выходе из `main`. В одном блоке автоматически созданные объекты уничтожаются в порядке, обратном завершению их создания. Деструкторы вызываются также при досрочном выходе через `return`, если выполнение покидает область соответствующего объекта.

Деструктор выполняет завершающие действия объекта. Освобождение памяти, в которой лежит сам объект, зависит от способа её получения. Например, память локального объекта относится к его автоматическому хранению. Правила деструкторов: N4659, class.dtor.

## Управление ресурсом через время жизни: RAII

**RAII** связывает владение ресурсом со временем жизни объекта: получение ресурса оформляется при инициализации, освобождение — при уничтожении. Ресурсом может быть буфер памяти, открытый файл или захваченная блокировка.

Стандартный класс `std::vector` управляет динамическим массивом:

```
#include <vector>

int main()
{
    {
        std::vector<int> samples = {240, 250, 260, -1, 280};
        samples[0] = 245;
    }
    return 0;
}
```

`std::vector<int>` — вариант этого класса для элементов `int`; запись `<int>` задаёт параметр типа. При выходе из блока деструктор `samples` уничтожает элементы и освобождает память, которой владеет контейнер. Правила его создания и уничтожения: N4659, vector.cons.

## Владение динамическим массивом

Операция `new T` создаёт один объект, а `new T[n]` — массив из `n` объектов. Для обычного выделения памяти при невозможности получить блок возбуждается исключение `std::bad_alloc`. Заголовок `<new>` предоставляет объявление этого типа. Запись `new int[n]{}` инициализирует все элементы нулями; `new int[n]` оставляет значения элементов неопределёнными до записи. C++17: new-expression.

Способ освобождения должен соответствовать способу создания:

| Создание | Освобождение |
|---|---|
| `new T` | `delete pointer` |
| `new T[n]` | `delete[] pointer` |

Таблица 16: Владение динамическим массивом

`delete[]` уничтожает элементы массива и освобождает его память. Нельзя заменять его на `delete` или `free`; память, полученная через `malloc`, освобождают через `free`. C++17: delete-expression.

В классе, владеющем массивом, указатель и размер входят в состояние объекта. Конструктор получает память, деструктор освобождает её. Следующая программа предполагает положительный размер, заданный вызывающим кодом:

```
#include <iostream>
#include <new>

class IntBuffer {
public:
    explicit IntBuffer(int size)
        : size_(size), data_(new int[size]{})
    {
    }

    ~IntBuffer()
    {
        delete[] data_;
    }
```

```
    IntBuffer(const IntBuffer&) = delete;
    IntBuffer& operator=(const IntBuffer&) = delete;

    int size() const
    {
        return size_;
    }

    bool set(int index, int value)
    {
        if (index < 0 || index >= size_) {
            return false;
        }
        data_[index] = value;
        return true;
    }

    bool get(int index, int& value) const
    {
        if (index < 0 || index >= size_) {
            return false;
        }
        value = data_[index];
        return true;
    }

private:
    int size_;
    int* data_;
};

int main()
{
    try {
        IntBuffer values(3);
        values.set(1, 270);
        int result = 0;
        if (values.get(1, result)) {
            std::cout << result << '\n';
        }
    } catch (const std::bad_alloc&) {
        std::cout << "MEMORY ERROR\n";
        return 1;
    }
    return 0;
}
```

При успешном выделении программа выводит 270. Неудачный `set` сохраняет массив, неудачный `get` сохраняет выходной параметр. Ссылка `int& value` предоставляет место для результата; возвращаемый `bool` обозначает успех операции. Методы чтения имеют квалификатор `const`.

Обычное копирование поля-указателя повторяет адрес. Если разрешить такое копирование `IntBuffer`, два объекта будут считать один массив своим: изменение данных станет общим, а деструкторы попытаются освободить один блок дважды. Поэтому копирующий конструктор и оператор копирующего присваивания объявлены с `= delete`. Эти объявления запрещают соответствующие операции при компиляции. Для независи-

мо владеющей копии потребовались бы отдельное выделение и копирование элементов. Перемещение в этом примере также не предоставляется.

`try` задаёт блок, исключение из которого может быть обработано соответствующим `catch`. При выходе из него из-за исключения уничтожаются уже полностью созданные автоматические объекты. Если конструктор `IntBuffer` не завершился, деструктор самого `IntBuffer` не вызывается; при отказе его единственного выделения массив не был получен. В конструкторах с несколькими ресурсами каждый уже полученный ресурс должен иметь собственное средство освобождения. C++17: уничтожение при обработке исключения.

Матрица может владеть одним плоским массивом из `rows * columns` элементов. При построчном размещении элемент `(row, column)` имеет индекс `row * columns + column`. До обращения проверяют обе координаты; произведение размеров должно быть представимо в используемом типе. Поля размеров и указателя закрыты, а доступ выполняется через методы класса.

## 8.4. Наследование и полиморфизм

### Общий интерфейс правил тревоги

Пусть одна программа должна анализировать температуры с разными правилами: достижение верхнего порога либо снижение до нижнего. Обход массива и вычисление среднего одинаковы; выбор тревожных измерений задаётся правилом.

**Наследование** позволяет определить производный класс на основе базового. **Динамический полиморфизм** позволяет через общий интерфейс вызвать реализацию, соответствующую конкретному типу объекта.

Зададим базовый класс:

```
class AlarmRule {
public:
    virtual bool is_alarm(int raw) const = 0;
    virtual ~AlarmRule() = default;
};
```

`virtual` задаёт виртуальный метод. Запись `= 0` делает его **чисто виртуальным**: интерфейс требует реализации от конкретного производного класса. `AlarmRule` — **абстрактный класс**; создать самостоятельный объект `AlarmRule rule;` нельзя, но можно использовать указатели и ссылки на него. Правила абстрактных классов: N4659, class.abstract.

`virtual ~AlarmRule()` задаёт виртуальный деструктор. Динамически созданный через `new` объект уничтожают через `delete`: оператор вызывает деструктор и освобождает выделенную память. Виртуальный деструктор необходим для корректного уничтожения производного объекта через указатель на базовый. `= default` поручает компилятору сформировать реализацию деструктора. Правила удаления: N4659, expr.delete.

Теперь заменим прежний `ThresholdAlarm` версией с общим интерфейсом:

```
class ThresholdAlarm : public AlarmRule {
public:
    explicit ThresholdAlarm(int threshold_raw)
        : threshold_raw_(threshold_raw)
    {
    }

    bool is_alarm(int raw) const override
    {
        return raw >= threshold_raw_;
    }

private:
    int threshold_raw_;
};
```

`public AlarmRule` означает публичное наследование. Объект `ThresholdAlarm` можно использовать через доступный интерфейс `AlarmRule`. Его базовая часть инициализируется до полей производного класса; при уничтожении порядок обратный: сначала завершается производная часть, затем базовая.

`override` требует, чтобы метод переопределял виртуальный метод базового класса. Например, случайный пропуск `const` изменил бы сигнатуру; с `override` компилятор обнаружит несовпадение. Закрытые поля базового класса, если они есть, не становятся напрямую доступными методам производного.

Второе правило имеет другой способ сравнения:

```
class LowerThresholdAlarm : public AlarmRule {
public:
    explicit LowerThresholdAlarm(int threshold_raw)
        : threshold_raw_(threshold_raw)
    {
    }

    bool is_alarm(int raw) const override
    {
        return raw <= threshold_raw_;
    }

private:
    int threshold_raw_;
};
```

## Вызов через ссылку на базовый класс

```
ThresholdAlarm upper(270);
LowerThresholdAlarm lower(250);

const AlarmRule &first_rule = upper;
const AlarmRule &second_rule = lower;

bool high_alarm = first_rule.is_alarm(280);
bool low_alarm = second_rule.is_alarm(240);
```

Оба результата равны `true`. Статический тип обеих ссылок — `const AlarmRule &`;

конкретные объекты имеют типы `ThresholdAlarm` и `LowerThresholdAlarm`. Виртуальный вызов выбирает соответствующее переопределение. Ссылка обозначает существующий объект и сохраняет возможность такого вызова; она не создаёт копию его базовой части. Правила виртуальных функций: N4659, class.virtual.

Вызов через общий интерфейс полезен, когда код обработки должен работать с несколькими реализациями. Прямой вызов метода конкретного класса подходит для работы с одним известным типом.

## Владение объектами через std::unique_ptr

`std::unique_ptr<T>` из `<memory>` владеет динамически созданным объектом. При уничтожении владельца объект освобождается; копирование владельца запрещено. `get()` возвращает невладеющий указатель, `*pointer` обозначает объект, а `pointer->method()` вызывает его метод. Эти обращения требуют существующего объекта. C++17: unique_ptr.

`std::make_unique<T>(arguments)` создаёт объект типа `T`, передаёт аргументы его конструктору и возвращает `unique_ptr<T>`. Для массива применяется отдельный вариант `unique_ptr<T[]>`; он использует освобождение через `delete[]`. C++17: make_unique.

Разные производные объекты можно хранить через владельцев одного базового типа:

```
#include <iostream>
#include <memory>
#include <vector>

class AlarmRule {
public:
    virtual ~AlarmRule() = default;
    virtual bool is_alarm(int value) const = 0;
};

class AboveThreshold : public AlarmRule {
public:
    explicit AboveThreshold(int threshold) : threshold_(threshold) {}

    bool is_alarm(int value) const override
    {
        return value >= threshold_;
    }

private:
    int threshold_;
};

class OutsideRange : public AlarmRule {
public:
    OutsideRange(int lower, int upper) : lower_(lower), upper_(upper) {}

    bool is_alarm(int value) const override
    {
        return value < lower_ || value > upper_;
    }
```

```
private:
    int lower_;
    int upper_;
};

int main()
{
    std::vector<std::unique_ptr<AlarmRule>> rules;
    rules.push_back(std::make_unique<AboveThreshold>(270));
    rules.push_back(std::make_unique<OutsideRange>(200, 300));

    for (const auto& rule : rules) {
        std::cout << (rule->is_alarm(280) ? "YES" : "NO") << '\n';
    }
    return 0;
}
```

Программа выводит `YES` и `NO`. Контейнер хранит владельцев, а сами объекты имеют разные производные типы. Временный результат `make_unique` передаёт владение элементу контейнера. `const auto&` обозначает ссылку на существующий элемент с автоматически выведенным типом; копирование `unique_ptr` при обходе не выполняется. Квалификатор `const` у владельца сам по себе не делает объект по указателю константным.

При уничтожении `rules` уничтожаются его элементы и принадлежащие им объекты. Виртуальный деструктор `AlarmRule` обеспечивает уничтожение производного объекта через базовый тип. Вызов `is_alarm` выбирает переопределение по типу объекта; проверка типа и приведение к производным классам здесь не требуются.

Из именованного владельца ресурс передают с помощью `std::move` из `<utility>`. Например, после `auto second = std::move(first);` для двух владельцев совместимого типа ресурс принадлежит `second`, а `first` пуст. `std::move` делает возможным выбор операции перемещения; сама передача выполняется этой операцией. Невладеющий указатель из `get()` не продлевает время жизни объекта, и через него нельзя повторно освобождать ресурс.

## 8.5. Пример: один анализатор и разные правила тревоги

В каталоге `examples/lab00_oop` заголовок `alarm_rules.hpp` содержит `AlarmRule` и обе производные реализации из предыдущего раздела, с защитой от повторного включения. Файл `main.cpp` использует их:

```
#include <cstddef>
#include "alarm_rules.hpp"

struct TemperatureStats {
    std::size_t valid_count = 0;
    std::size_t alarm_count = 0;
    double average_c = 0.0;
};

TemperatureStats analyse_temperatures(const int *raw, std::size_t count,
                                      const AlarmRule &rule)
{
```

```
    TemperatureStats result;
    double sum_c = 0.0;

    for (std::size_t i = 0; i < count; ++i) {
        if (raw[i] < 0 || raw[i] > 500) {
            continue;
        }
        ++result.valid_count;
        sum_c += raw[i] / 10.0;
        if (rule.is_alarm(raw[i])) {
            ++result.alarm_count;
        }
    }

    if (result.valid_count != 0) {
        result.average_c = sum_c / result.valid_count;
    }
    return result;
}

int main()
{
    const int raw[] = {240, 250, 260, -1, 280};
    std::size_t count = sizeof raw / sizeof raw[0];
    ThresholdAlarm upper(270);
    LowerThresholdAlarm lower(250);

    TemperatureStats upper_stats = analyse_temperatures(raw, count, upper);
    TemperatureStats lower_stats = analyse_temperatures(raw, count, lower);

    if (upper_stats.valid_count == 0 || lower_stats.valid_count == 0) {
        return 1;
    }
    return 0;
}
```

`std::size_t` из `<cstddef>` выполняет ту же роль типа размера, что `size_t` в C. Начальные значения полей заданы в определении структуры, поэтому `TemperatureStats result;` создаёт результат с нулевыми счётчиками и средним.

Как и при передаче массива в C, `raw` должен указывать на доступные `count` элементов. При `count == 0` допускается нулевой указатель. В этом варианте признаком наличия данных служит `valid_count > 0`; при нулевом количестве допустимых измерений значение `average_c` не используют как результат измерения.

Функция исключает недопустимые измерения до вызова правила и накапливает общие показатели. Параметр `const AlarmRule &rule` позволяет передать любую из двух реализаций. Для первого вызова тревожным является 280, для второго — 240 и 250.

| Поле | `upper_stats` | `lower_stats` |
|---|---|---|
| `valid_count` | 4 | 4 |
| `alarm_count` | 1 | 2 |
| `average_c` | 25.75 | 25.75 |

Таблица 17: 8.5. Пример: один анализатор и разные правила тревоги

Для сборки из каталога проекта:

```
g++ -std=c++17 -Wall -Wextra -g -O0 main.cpp -o temperature_oop.exe
```

Поставьте точку останова на `return 0;` и исследуйте обе структуры. Измените пороги, предварительно рассчитав ожидаемое число тревог, затем сопоставьте его с результатом выполнения.

Класс полезен, когда данные и операции образуют устойчивую ответственность: допустимая температура, правило сравнения, владение буфером. При проектировании выбирают допустимые состояния, публичные операции и способы обработки ошибок. Публичное наследование задаёт возможность использовать производный объект через интерфейс базового.

## 8.6. Перегрузка функций

**Перегрузка** позволяет определить несколько функций с одним именем и разными списками параметров. Компилятор выбирает подходящую функцию по аргументам вызова. Например, одну операцию ограничения значения можно предоставить для целых и вещественных чисел:

```
#include <iostream>

int clamp_value(int value, int lower, int upper)
{
    if (value < lower) {
        return lower;
    }
    if (value > upper) {
        return upper;
    }
    return value;
}

double clamp_value(double value, double lower, double upper)
{
    if (value < lower) {
        return lower;
    }
    if (value > upper) {
        return upper;
    }
    return value;
}

int main()
{
    std::cout << clamp_value(120, 0, 100) << '\n';
    std::cout << clamp_value(27.5, 0.0, 50.0) << '\n';
    return 0;
}
```

Вывод — две строки: `100` и `27.5`. В первом вызове аргументы имеют тип `int`, во втором — `double`; в обоих случаях существует перегрузка с точным соответствием типов. Для обеих функций предполагается `lower` $\leq$ `upper`.

Перегрузки могут отличаться количеством, типами или порядком типов параметров. Различие только в именах параметров или возвращаемом типе не создаёт перегрузку. Например, объявления `int read_value()` и `double read_value()` в одной области недопустимы. C++17: перегружаемые объявления.

При выборе учитываются и допустимые преобразования аргументов. Если нельзя определить единственную лучшую функцию, вызов неоднозначен и программа не компилируется. В данном примере `clamp_value(27.5, 0, 50)` неоднозначен: вещественная перегрузка лучше соответствует первому аргументу, а целочисленная — двум остальным. Запись границ `0.0` и `50.0` устраняет неоднозначность и явно задаёт вещественные вычисления.

Методы класса и конструкторы также могут перегружаться. Перегрузка и переопределение виртуального метода решают разные задачи: перегрузка предлагает несколько списков параметров, а переопределение задаёт реализацию виртуальной операции в производном классе. Выбор перегрузки определяется при компиляции; виртуальный вызов может затем выбрать реализацию по фактическому типу объекта.

## 8.7. Шаблоны функций и классов

В двух перегрузках `clamp_value` алгоритм одинаков, различаются типы. **Шаблон** описывает семейство функций или классов через параметры, которые задаются при его использовании. Параметр типа позволяет один раз записать обработку, подходящую для нескольких типов.

### Шаблон функции

```
#include <iostream>

template <typename T>
T clamp_value(T value, T lower, T upper)
{
    if (value < lower) {
        return lower;
    }
    if (value > upper) {
        return upper;
    }
    return value;
}

int main()
{
    std::cout << clamp_value(120, 0, 100) << '\n';
    std::cout << clamp_value(27.5, 0.0, 50.0) << '\n';
    std::cout << clamp_value<double>(27.5, 0, 50) << '\n';
    return 0;
}
```

Этот пример заменяет две предыдущие перегрузки одним шаблоном. Вывод: `100`, `27.5` и `27.5`, каждое число с новой строки.

В записи `template <typename T>` имя `T` обозначает параметр типа. Оно используется как тип результата, параметров и локальных переменных. В этом объявлении вместо

`typename` допустимо `class`; это не ограничивает параметр только классовыми типами. C++17: шаблоны.

Для первого вызова компилятор выводит `T = int`, для второго — `T = double`. Получение конкретного варианта из шаблона называется **инстанцированием**. Тип выбирается при компиляции; переменная `T` во время выполнения не существует.

Вызов `clamp_value(27.5, 0, 50)` для этого шаблона не компилируется: из первого аргумента выводится `double`, а из остальных — `int`, хотя параметр `T` один. Вывод параметра типа здесь не выбирает общий арифметический тип автоматически. В третьем вызове тип задан явно: `clamp_value<double>(...)`. После этого целые границы преобразуются в `double`. C++17: вывод параметров шаблона при вызове.

Тип должен поддерживать операции, использованные в шаблоне. Для `clamp_value` нужны сравнения и возможность вернуть значение по значению; числа `int` и `double` этим требованиям соответствуют. Для вещественного варианта в примерах предполагаются обычные конечные значения. Шаблон не делает произвольный тип пригодным для любого алгоритма.

## Шаблон класса

Шаблон класса позволяет хранить и обрабатывать данные разных типов через одинаковый интерфейс:

```
#include <iostream>

template <typename T>
class Range {
public:
    Range(T lower, T upper)
        : lower_(lower), upper_(upper)
    {
    }

    bool contains(T value) const
    {
        return value >= lower_ && value <= upper_;
    }

private:
    T lower_;
    T upper_;
};

int main()
{
    Range<int> raw_range(0, 500);
    Range<double> celsius_range(0.0, 50.0);

    std::cout << std::boolalpha;
    std::cout << raw_range.contains(250) << '\n';
    std::cout << celsius_range.contains(50.5) << '\n';
    return 0;
}
```

Вывод: `true` и `false`. `Range<int>` и `Range<double>` — разные типы, созданные из

одного шаблона. В первом поля имеют тип `int`, во втором — `double`. Конструктор получает границы, метод проверяет принадлежность диапазону. Вызывающий код должен передавать границы в порядке `lower` $\leq$ `upper`.

Ранее использованный `std::vector<int>` устроен по тому же принципу: `std::vector` — шаблон класса, а `int` задаёт тип его элементов. Для `Range<double>` и `std::vector<double>` также используется параметр типа `double`, но эти классы имеют разные назначения и интерфейсы.

### Шаблоны в многофайловом проекте

При обычном неявном инстанцировании используемое определение шаблона должно быть доступно в единице трансляции, где требуется конкретный вариант. Поэтому определения шаблонов функций и классов обычно размещают в заголовке, который подключает пользователь. Одного объявления шаблона в `.hpp` и его определения в отдельном `.cpp` для такого использования недостаточно. Существуют способы явно инстанцировать выбранные варианты в `.cpp`; они выходят за базовое знакомство с шаблонами.

Перегрузка полезна, когда для разных параметров нужны разные реализации одной операции. Шаблон полезен, когда одинаковый алгоритм или устройство класса подходит нескольким типам. Эти механизмы относятся к C++ и могут использоваться как вместе с классами, так и для свободных функций.

## 8.8. Стек

**Стек** — структура данных, в которой последний добавленный элемент извлекается первым: LIFO (*last in, first out*). Операция `push` добавляет элемент на вершину, `pop` извлекает вершину, `top` читает её без удаления. `size` сообщает количество элементов, `empty` — отсутствие элементов.

Стек фиксированной вместимости можно представить массивом и счётчиком. Если в стеке `size_` элементов, они занимают позиции `0..size_ - 1`; вершина находится в `size_ - 1`, а новая запись выполняется в `size_`. Инвариант: `0` $\leq$ `size_` $\leq$ `capacity`. Добавление в полный стек и извлечение из пустого отклоняются до обращения к массиву.

```
#include <iostream>

class IntStack {
public:
    bool push(int value)
    {
        if (size_ == capacity) {
            return false;
        }
        data_[size_++] = value;
        return true;
    }

    bool pop(int& value)
    {
        if (empty()) {
            return false;
        }
```

```
        value = data_[--size_];
        return true;
    }

    bool top(int& value) const
    {
        if (empty()) {
            return false;
        }
        value = data_[size_ - 1];
        return true;
    }

    int size() const { return size_; }
    bool empty() const { return size_ == 0; }

private:
    static constexpr int capacity = 3;
    int data_[capacity];
    int size_ = 0;
};

int main()
{
    IntStack values;
    values.push(10);
    values.push(20);
    int value = 0;
    while (values.pop(value)) {
        std::cout << value << '\n';
    }
    return 0;
}
```

Вывод: `20`, затем `10`. При отказе `pop` или `top` выходной параметр сохраняется. `capacity` — статическая константа класса, общая для его объектов; `data_` и `size_` принадлежат каждому объекту отдельно. Инициализировать весь массив не требуется: методы читают только ранее добавленные элементы.

В этой реализации число действий каждой операции не зависит от количества элементов. Такую временную сложность обозначают `O(1)`. Обход всех `n` элементов имеет сложность `O(n)`; обозначение характеризует рост числа действий при увеличении входных данных, а не точное время исполнения.

Для проверки скобок стек хранит открывающие символы. Открывающая скобка помещается в стек; закрывающая должна соответствовать его вершине и удаляет её. Пустой стек при закрывающей скобке или несовпадение видов означает ошибку. После обработки всей строки стек должен быть пуст. Поэтому `([])` корректна, а `([)]` — нет, хотя количества открывающих и закрывающих скобок совпадают. Просмотр строки выполняется за `O(n)`, вместимость стека должна учитывать максимально возможное число незакрытых скобок.

`std::string` из `<string>` хранит последовательность символов и управляет её памятью. `std::cin >> text` читает слово до пробельного символа; для скобочной строки без пробелов этого достаточно. Цикл `for (char symbol : text)` последовательно получает её символы. Контейнер строки предоставляет входные данные; стек задаёт порядок

проверки вложенности.

## 8.9. Очередь и кольцевой буфер

**Очередь** извлекает элементы в порядке добавления: FIFO (*first in, first out*). `push` добавляет элемент в конец, `pop` извлекает первый, `front` читает первый без удаления. Перенос всех оставшихся элементов массива после каждого извлечения потребовал бы `O(n)` действий.

**Кольцевая очередь** использует массив фиксированной вместимости `capacity`, индекс первого элемента `head` и количество элементов `size`. Логическому элементу с индексом `i` соответствует физическая позиция `(head + i) % capacity`. Позиция следующей записи: `(head + size) % capacity`. Вместимость положительна; $0 \le$ `head` $<$ `capacity`, $0 \le$ `size` $\le$ `capacity`.

```
#include <iostream>

class IntQueue {
public:
    bool push(int value)
    {
        if (size_ == capacity) {
            return false;
        }
        data_[(head_ + size_) % capacity] = value;
        ++size_;
        return true;
    }

    bool pop(int& value)
    {
        if (empty()) {
            return false;
        }
        value = data_[head_];
        head_ = (head_ + 1) % capacity;
        --size_;
        return true;
    }

    bool front(int& value) const
    {
        if (empty()) {
            return false;
        }
        value = data_[head_];
        return true;
    }

    int size() const { return size_; }
    bool empty() const { return size_ == 0; }

private:
    static constexpr int capacity = 3;
    int data_[capacity];
    int head_ = 0;
    int size_ = 0;
```

```
};

int main()
{
    IntQueue values;
    values.push(10);
    values.push(20);
    values.push(30);
    int value = 0;
    values.pop(value);
    std::cout << value << '\n';
    values.push(40);
    while (values.pop(value)) {
        std::cout << value << '\n';
    }
    return 0;
}
```

Вывод: `10`, `20`, `30`, `40`, каждое число с новой строки. После извлечения `10` запись `40` использует освободившуюся позицию `0`. Физическое размещение в массиве отличается от логического порядка. Счётчик `size_` позволяет различить пустое и полное состояния; сдвиг элементов не выполняется. Операции имеют сложность `O(1)`.

### Буфер последних измерений

Кольцевой буфер может использовать другую политику заполнения: при добавлении в полный буфер старейший элемент заменяется новым. Если буфер ещё не заполнен, запись выполняется в `(head + size) % capacity`, после чего размер увеличивается. Если заполнен, запись выполняется в `head`, а `head` продвигается на одну позицию; размер сохраняется.

Пусть вместимость равна трём. После добавления `10, 20, 30, 40, 50` буфер содержит `30, 40, 50` в порядке от старого к новому. Чтение элемента `i` выполняется по той же формуле `(head + i) % capacity`, где `0` $\leq$ `i < size`.

Среднее можно вычислять за `O(1)`, поддерживая сумму: при вытеснении сначала вычитают старое значение, затем прибавляют новое. Для целочисленных измерений тип суммы выбирают с учётом максимального размера и диапазона значений; в заданиях используется `long long`. Деление выполняют после преобразования суммы в `double`. Среднее пустого буфера определяется контрактом класса; в задании оно равно `0.0`.

## 8.10. Стандартные контейнеры C++

**Контейнер** хранит элементы и предоставляет операции над ними. Стандартные контейнеры владеют своим хранилищем; память освобождается при уничтожении. Тип элемента задаётся параметром шаблона. Выбор контейнера определяется требуемыми операциями и способом доступа к данным.

### std::vector: последовательность переменного размера

`std::vector<T>` из `<vector>` хранит элементы последовательно в непрерывной области памяти. У `std::vector<int> values` исходно нет элементов. `size()` возвращает их количество, `empty()` проверяет пустоту. `push_back(value)` добавляет элемент в конец,

`pop_back()` удаляет последний и требует непустого контейнера. `values[index]` предоставляет элемент и требует `index < values.size()`; автоматической проверки границ этот оператор не выполняет. C++17: vector.

Для позиции используется **итератор** — объект, обозначающий элемент последовательности. `begin()` обозначает начало, `end()` — позицию за последним элементом, которую нельзя разыменовывать. Для `vector` выражение `begin() + index` задаёт позицию по индексу. Вставка допускает позицию `end()`, удаление отдельного элемента — только позицию существующего элемента.

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> values;
    values.push_back(10);
    values.push_back(30);
    values.insert(values.begin() + 1, 20);
    values.erase(values.begin());

    for (int value : values) {
        std::cout << value << '\n';
    }
    std::cout << values.size() << '\n';
    return 0;
}
```

Вывод: 20, 30, 2. Вставка выполняется перед указанной позицией, удаление сдвигает последующие элементы. Отрицательный индекс проверяют до преобразования в беззнаковый тип размера. Для вставки допустимы индексы `0..size`, для чтения и удаления — `0..size - 1`.

`capacity()` сообщает вместимость выделенного хранилища, `size()` — число существующих элементов. `reserve(n)` резервирует место, не создавая элементы; `resize(n)` изменяет их количество. Увеличение вместимости может переместить элементы в другой блок. Тогда прежние указатели, ссылки и итераторы на элементы становятся недействительными. Даже без такого перемещения вставка делает недействительными позиции начиная с места вставки; удаление — начиная с места удаления. После изменения контейнера позиции для последующих операций вычисляют заново. C++17: модификации vector.

Доступ по индексу занимает `O(1)`. Добавление в конец имеет амортизированную сложность `O(1)`: отдельное добавление с перевыделением может занять `O(n)`, но стоимость серии добавлений распределяется по операциям. Вставка и удаление в середине требуют перемещения последующих элементов и имеют сложность `O(n)`.

## std::map: соответствие ключа и значения

`std::map<Key, Value>` из `<map>` хранит пары «ключ — значение» с уникальными ключами. Для `std::map<int, int>` с обычным сравнением обход идёт по возрастанию ключей. `size()` считает пары, `find(key)` возвращает итератор найденной пары либо `end()`. У пары `first` обозначает ключ, `second` — связанное значение. Поиск, вставка и удаление по ключу имеют логарифмическую сложность `O(log n)`. C++17: требования

к ассоциативным контейнерам.

`counts[key]` возвращает ссылку на значение. Если ключ отсутствует, создаётся новая запись; для значения типа `int` её начальное значение равно нулю. Поэтому `++counts[value]` подходит для подсчёта частот. Для проверки наличия без изменения контейнера используется `find`, а не `operator[]`. C++17: доступ к элементам map.

```
#include <iostream>
#include <map>

int main()
{
    const int samples[] = {270, 250, 270, 280, 250};
    std::map<int, int> counts;
    for (int value : samples) {
        ++counts[value];
    }
    for (const auto& entry : counts) {
        std::cout << entry.first << ' ' << entry.second << '\n';
    }
    return 0;
}
```

Вывод:

```
250 2
270 2
280 1
```

`const auto& entry` позволяет читать пару без копирования. `erase(key)` удаляет запись по ключу и возвращает число удалённых записей: для `map` оно равно нулю или единице. Ключ существующей записи не изменяют через итератор; чтобы заменить ключ, старую запись удаляют и добавляют новую.

## std::set: множество уникальных значений

`std::set<T>` из `<set>` хранит уникальные значения. Для `std::set<int>` с обычным сравнением обход идёт по возрастанию. `insert(value)` возвращает пару: `first` — итератор соответствующего элемента, `second` — признак фактического добавления. При повторном добавлении `second == false`. `erase(value)` возвращает число удалённых элементов, `count(value)` — ноль или единицу. C++17: set.

```cpp
#include <iostream>
#include <set>

int main()
{
    std::set<int> identifiers;
    identifiers.insert(7);
    identifiers.insert(3);
    bool added = identifiers.insert(7).second;
    std::cout << (added ? "ADDED" : "EXISTS") << '\n';
    std::cout << identifiers.count(3) << '\n';
    identifiers.erase(7);
    for (int value : identifiers) {
        std::cout << value << '\n';
    }
    return 0;
}
```

Вывод: `EXISTS`, `1`, `3`. У множества нет доступа по числовому индексу. Изменять значение через итератор нельзя: оно определяет положение элемента в порядке контейнера. Поиск и изменение по значению имеют сложность `O(log n)`.

| Тип | Организация | Основное применение |
|---|---|---|
| Стек | Доступ к последнему добавленному элементу | Вложенность, скобочные последовательности |
| Очередь | Извлечение в порядке поступления | Последовательная обработка команд |
| Кольцевой буфер с вытеснением | Сохранение последних элементов | История измерений ограниченного размера |
| `std::vector<T>` | Последовательность с доступом по индексу | Изменяемый список |
| `std::map<Key, Value>` | Уникальный ключ и связанное значение | Частоты, поиск записи по ключу |
| `std::set<T>` | Уникальные упорядоченные значения | Регистрация идентификаторов |

Таблица 18: std::set: множество уникальных значений

При собственной реализации структуры класс задаёт интерфейс, скрывает хранилище и поддерживает инварианты. При использовании стандартного контейнера эти обязанности выполняет библиотечный класс; вызывающий код соблюдает условия его операций.

## 8.11. Контрольные вопросы

1. Чем класс отличается от объекта этого класса? Что независимо хранится у двух объектов `Temperature`?
2. Для какого объекта выполняется вызов `first.celsius()`? Какова роль `this`?
3. Что означает `const` после списка параметров метода? Почему через `const Temperature &` нельзя вызвать `set_raw`?
4. Чем отличаются `public`, `private` и `protected`? Какой доступ имеют члены `class` и `struct` по умолчанию?

5. Какое условие поддерживает вторая версия `Temperature`? Что происходит после неудачного `set_raw(-1)`?
6. Почему одного закрытого поля недостаточно для корректного класса? Какие обязанности остаются у его методов?
7. Что происходит при `Temperature copy = sample` в данном примере? Влияет ли изменение копии на оригинал?
8. Когда вызывается конструктор и чем его запись отличается от обычного метода?
9. Чем список инициализации отличается от присваивания в теле конструктора? В каком порядке инициализируются поля?
10. Для чего здесь используется `explicit`? Почему нельзя создать `ThresholdAlarm` без указания порога?
11. Когда уничтожаются `inner` и `outer` в примере `ScopeTrace`? Как изменится порядок при досрочном `return` из внутреннего блока?
12. Чем действия деструктора отличаются от освобождения памяти самого объекта?
13. Как RAII связывает ресурс со временем жизни объекта? Что освобождает деструктор `std::vector`?
14. Почему `AlarmRule` абстрактный? Для чего можно использовать этот тип, если отдельный объект создать нельзя?
15. Что дают `virtual` и `override`? Что обнаружит компилятор при пропуске `const` у переопределения с `override`?
16. Почему виртуальный деструктор важен при уничтожении производного объекта через указатель на базовый?
17. Как выбирается метод при вызове `first_rule.is_alarm(280)` через ссылку на `AlarmRule`?
18. Почему оба анализа дают одинаковое среднее и разное число тревог? В каком месте исключается `-1`?
19. Как добавить третье правило, сохранив функцию `analyse_temperatures`? Какие операции обязано предоставить правило?
20. Для чего нужен namespace? Чем sensors::Temperature отличается от simulation:: Temperature и от обращения measured.celsius()?
21. Чем using std::cout отличается от using namespace std? Почему в заголовках проекта предпочтительны явные имена?
22. Заменяет ли пространство имён класс, объект или ограничения доступа public/private?
23. Какие различия позволяют перегрузить функцию? Почему различия только в возвращаемом типе недостаточно?
24. Почему вызов clamp_value(27.5, 0, 50) неоднозначен для двух перегрузок из раздела 8.6?
25. Чем перегрузка отличается от переопределения виртуального метода?
26. Что обозначает T в template <typename T>? Когда определяется конкретный тип?
27. Почему вывод типа не работает для clamp_value(27.5, 0, 50) из раздела 8.7? Что меняет явная запись clamp_value<double>?
28. Чем Range<int> отличается от Range<double>? Какие операции должен поддерживать параметр типа?
29. Почему определения шаблонов обычно находятся в заголовках? Когда удобнее перегрузка, а когда шаблон?
30. Чем отличаются `new T`, `new T[n]` и `new int[n]{}`? Какие операции освобождения им соответствуют?
31. Почему копирование одного указателя не создаёт независимо владеющую копию массива? Что запрещают два объявления с `= delete` в `IntBuffer`?

32. Как `get(index, value)` сообщает об отказе? Почему выходной параметр при отказе сохраняется?
33. Вызывается ли деструктор объекта, если его конструктор не завершился? Что происходит с уже полностью созданными автоматическими объектами при обработке исключения?
34. Как двумерные координаты преобразуются в индекс плоского хранилища матрицы? Какие границы проверяются?
35. Кто владеет объектом при использовании `unique_ptr`? Что возвращает `make_unique` и что означает указатель из `get()`?
36. Зачем в контейнере `unique_ptr<AlarmRule>` нужен виртуальный деструктор базового класса? Почему владельцы при обходе передаются по ссылке?
37. Как передаётся владение из именованного `unique_ptr` и в каком состоянии остаётся прежний владелец?
38. Как различаются LIFO и FIFO? Какие элементы читают `top` и `front`?
39. Как счётчик стека определяет место записи и индекс вершины? Какие условия проверяют перед добавлением и извлечением?
40. Почему подсчёта скобок недостаточно для проверки вложенности? Как стек обнаруживает ошибку в `([)]`?
41. Как вычисляются физическая позиция элемента и место следующей записи кольцевой очереди? Зачем хранить её размер?
42. Чем заполненная очередь отличается от заполненного буфера последних измерений при добавлении нового значения?
43. Как обновлять сумму при вытеснении старого измерения? Почему тип суммы может отличаться от типа элемента?
44. Чем различаются `size`, `capacity`, `reserve` и `resize` у `vector`? Какие позиции допустимы для вставки и удаления?
45. Когда указатели, ссылки и итераторы на элементы `vector` становятся недействительными?
46. Чем амортизированное `O(1)` добавление в конец отличается от сложности отдельного добавления с перевыделением памяти?
47. Почему `map::operator[]` может изменить контейнер при отсутствующем ключе? Как проверить наличие без добавления?
48. Что обозначают `first` и `second` у пары из `map` и у результата `set::insert`?
49. В каком порядке обходятся `map<int, int>` и `set<int>` с обычным сравнением? Почему ключи нельзя изменять через итератор?

# Практическая часть

Задания глав 1–3 выполняются на C11 или C++17. Дополнительная глава 8 предназначена только для C++17. Для каждой задачи отправляйте на сайт SDB Lab полную программу с функцией `main`, включая требуемые функции и определения типов. Программа читает данные из стандартного ввода и записывает ответ в стандартный вывод. Дополнительные сообщения, приглашения к вводу и отладочный вывод не допускаются.

Требования к алгоритмам, интерфейсам функций и динамической памяти приведены в условиях. Входные данные в примерах соответствуют ограничениям задач. Пустой вывод и пустая строка отмечены явно.

## Глава 1

### Блок 1.1. Целочисленные вычисления

#### 1.1.1. Разность, произведение и сумма

С клавиатуры вводятся два целых числа `a` и `b`. Напишите программу, которая вычислит и выведет на отдельных строках их разность, произведение и сумму в указанном порядке. Разность вычисляется как `a − b`. Для вычислений используйте тип `long long`, поскольку результаты могут не помещаться в 32-битный `int`.

**Формат входных данных**

Два целых числа `a` и `b`, каждое с новой строки. Оба числа находятся в диапазоне 32-битного знакового `int`: $-2^{31} \le$ `a, b` $\le 2^{31} - 1$.

**Формат выходных данных**

Три целых числа, каждое с новой строки: разность `a − b`, произведение `a · b` и сумма `a + b`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>2 | 3<br>10<br>7 |
| -3<br>4 | -7<br>-12<br>1 |
| 2147483647<br>-2147483648 | 4294967295<br>-4611686016279904256<br>-1 |

#### 1.1.2. Значение выражения с квадратом

Дано целое число `x`. Вычислите значение выражения $x^2$ `+ 3x − 5` и выведите результат. Для вычислений используйте тип `long long`, поскольку результат может не помещаться в 32-битный `int`.

**Формат входных данных**

Одно целое число x в диапазоне 32-битного знакового int: $-2^{31} \le x \le 2^{31} - 1$.

**Формат выходных данных**

Одно целое число — значение выражения.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3 | 13 |
| -2 | -7 |
| 2147483647 | 4611686020574871545 |

### 1.1.3. Случайный прохожий

Случайный прохожий бежит к выходу из парка со скоростью v метров в минуту. Изначально расстояние до выхода равно x метров. Определите, сколько метров ему останется до выхода через m минут. Гарантируется, что за это время он не выбежит из парка.

**Формат входных данных**

Три целых положительных числа v, x и m, каждое с новой строки. Ограничения: $v < 100$, $x < 5000$, $m < 100$, $v \cdot m \le x$.

**Формат выходных данных**

Одно целое число — оставшееся расстояние до выхода из парка в метрах.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 10<br>150<br>10 | 50 |
| 5<br>100<br>20 | 0 |

## Блок 1.2. Деление, остаток и разряды числа

### 1.2.1. Закупка шоколада

Одна шоколадка стоит n рублей. У вас есть 562 рубля. Определите, сколько шоколадок вы можете купить и сколько рублей останется после покупки максимально возможного количества шоколадок.

**Формат входных данных**

Одно целое число n — стоимость шоколадки в рублях. Ограничение: $1 < n < 1000$.

**Формат выходных данных**

Два целых числа в одной строке через пробел: количество купленных шоколадок и оставшаяся сумма в рублях.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 50 | 11 12 |
| 600 | 0 562 |

## 1.2.2. Дни в неделе

С некоторого момента прошло x дней. Определите, сколько полных недель прошло за это время.

**Формат входных данных**

Одно целое число x — количество прошедших дней. Ограничение: $0 \le x \le 2^{31} - 1$.

**Формат выходных данных**

Одно целое число — количество полных недель.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 15 | 2 |
| 0 | 0 |

## 1.2.3. Электронные часы-1

С начала суток прошло x секунд. Определите количество полных часов с начала суток, полных минут с начала текущего часа и секунд с начала текущей минуты.

**Формат входных данных**

Одно целое число x — количество секунд, прошедших с начала суток. Ограничение: $0 \le x < 86400$.

**Формат выходных данных**

Три целых числа в одной строке через пробел: часы, минуты и секунды.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 7281 | 2 1 21 |
| 86399 | 23 59 59 |
| 0 | 0 0 0 |

### 1.2.4. День недели — 1

Дан порядковый номер дня года, считая с 1 января. Определите номер дня недели, если 1 января — понедельник. Использовать `if` и `while` запрещено.

**Формат входных данных**

Одно целое число `k` — порядковый номер дня года. Ограничение: $1 \le k \le 365$.

**Формат выходных данных**

Одно целое число — номер дня недели: понедельник — 1, вторник — 2, среда — 3, четверг — 4, пятница — 5, суббота — 6, воскресенье — 7.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 | 1 |
| 9 | 2 |

### 1.2.5. Цифры четырёхзначного числа

Напишите программу, которая определит цифры четырёхзначного числа `x` и выведет их в порядке от старшего разряда к младшему.

**Формат входных данных**

Одно целое число `x`. Ограничение: $1000 \le x \le 9999$.

**Формат выходных данных**

Четыре цифры, каждая с новой строки: тысячи, сотни, десятки и единицы числа `x`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1248 | 1<br>2<br>4<br>8 |
| 1005 | 1<br>0<br>0<br>5 |

### 1.2.6. Забытая цифра-3

При расчёте нагрузки для механизма подъёма грузов Архимед потерял одну цифру `x` и получил число `k`. Пропущенная цифра должна находиться на четвёртой позиции с конца в правильной записи числа. Напишите программу, которая вставит эту цифру в число `k` на указанную позицию, сохранив остальные цифры и их порядок.

**Формат входных данных**

Два целых числа, каждое с новой строки: сначала пропущенная цифра x, затем число k. Ограничения: $0 \le x \le 9$, $100 \le k \le 10^6$.

**Формат выходных данных**

Одно целое число — исправленный результат вычислений Архимеда.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>2714 | 23714 |
| 0<br>1234 | 10234 |
| 5<br>123 | 5123 |

## Блок 1.3. Вещественные вычисления

### 1.3.1. Верёвка

От верёвки длиной l отрезали три четверти её длины. Определите длину оставшейся части верёвки.

**Формат входных данных**

Одно вещественное число l — исходная длина верёвки. Ограничение: $0 < l \le 100$. Число содержит не более двух знаков после десятичной точки.

**Формат выходных данных**

Одно вещественное число — длина оставшейся части верёвки. Выведите результат не менее чем с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 99.50 | 24.875000 |
| 5.22 | 1.305000 |

### 1.3.2. На дачу!

Вычислите стоимость бензина для поездки на дачу и обратно. Расстояние до дачи в одну сторону составляет S километров. Автомобиль расходует B литров бензина на 100 километров пути. Один литр бензина стоит C рублей.

**Формат входных данных**

Три числа, каждое с новой строки: целое число S, затем вещественные числа B и C. Ограничения: $0 \le S \le 2^{31} - 1$, $0 \le B \le 10^5$, $0 \le C \le 10^5$. Числа B и C содержат не более двух знаков после десятичной точки.

**Формат выходных данных**

Одно вещественное число — стоимость поездки на дачу и обратно в рублях. Выведите результат с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 100<br>10.00<br>30.00 | 600.000000 |
| 200<br>7.00<br>50.00 | 1400.000000 |
| 0<br>7.00<br>50.00 | 0.000000 |

### 1.3.3. Пешеход — 2

Пешеход Василий обычно передвигается со скоростью `a` километров в час, а когда спешит — со скоростью `b` километров в час. Он вышел из дома и первые `t` часов шёл с обычной скоростью. Затем он понял, что опаздывает, и увеличил скорость до `b`. Расстояние от дома до цели составляет `c` километров. Определите общее время пути от выхода из дома до достижения цели.

**Формат входных данных**

Четыре вещественных числа `a`, `b`, `c` и `t`, каждое с новой строки. Ограничения: $0 < a, b, c, t \leq 20$, $a < b$, $a \cdot t < c$. Каждое число содержит не более одного знака после десятичной точки. Гарантируется, что Василий увеличил скорость до достижения цели.

**Формат выходных данных**

Одно вещественное число — общее время пути в часах с учётом изменения скорости. Выведите результат не менее чем с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5.0<br>7.5<br>15.0<br>1.5 | 2.500000 |
| 2.0<br>4.0<br>10.0<br>1.0 | 3.000000 |

## Блок 1.4. Условия и выбор

### 1.4.1. Обезьяна на ветке

Вы смотрите на дерево. Вводится целое число: 0 означает, что вы видите только дерево, а 1 — что на ветке сидит обезьяна. Напишите программу, которая выведет

соответствующие сообщения.

**Формат входных данных**

Одно целое число: 0 или 1.

**Формат выходных данных**

Если введено 1, выведите две строки:

```
Кажется, на ветке обезьяна
Не хочу больше смотреть на дерево
```

Если введено 0, выведите одну строку:

```
Не хочу больше смотреть на дерево
```

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 | Кажется, на ветке обезьяна<br>Не хочу больше смотреть на дерево |
| 0 | Не хочу больше смотреть на дерево |

## 1.4.2. Выбери телефон

Даны цены трёх мобильных телефонов. Захар хочет купить самый дорогой из них. Определите цену телефона, который он выберет.

**Формат входных данных**

Три целых числа `a`, `b` и `c` — цены телефонов. Ограничение: $0 \le a, b, c \le 10000$.

**Формат выходных данных**

Одно целое число — цена самого дорогого телефона.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3000 5500 10000 | 10000 |
| 8000 8000 1000 | 8000 |

## 1.4.3. Шахматная доска

Поле шахматной доски задаётся парой натуральных чисел от 1 до 8. Первое число — номер вертикали при счёте слева направо, второе — номер горизонтали при счёте снизу вверх. Определите, имеют ли поля `(a, b)` и `(c, d)` одинаковый цвет. Левое нижнее поле `(1, 1)` — чёрное.

**Формат входных данных**

Четыре натуральных числа a, b, c и d. Ограничение: $1 \le a, b, c, d \le 8$.

**Формат выходных данных**

Выведите YES, если поля одного цвета, и NO в противном случае.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 1 3 3 | YES |
| 2 2 4 5 | NO |

## 1.4.4. Счастливый билет — 1

Определите, является ли заданный номер билета счастливым. Счастливым называют шестизначный номер, у которого сумма первых трёх цифр равна сумме последних трёх цифр.

**Формат входных данных**

Одно натуральное число a — номер билета. Ограничение: $100000 \le a \le 999999$.

**Формат выходных данных**

Выведите YES, если номер билета счастливый, и NO в противном случае.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 421007 | YES |
| 123432 | NO |

## 1.4.5. Светофор

С начала каждого часа светофор работает по повторяющемуся циклу: три минуты горит зелёный сигнал, затем одну минуту — жёлтый, затем две минуты — красный. После этого цикл повторяется. По количеству минут t, прошедших с начала часа, определите цвет сигнала в этот момент. При смене сигналов в точности на границе интервала уже горит следующий сигнал.

**Для выбора цвета сигнала обязательно используйте оператор switch.**

**Формат входных данных**

Одно целое число t — время в минутах с начала часа. Ограничение: $0 \le t \le 59$.

**Формат выходных данных**

Выведите RED, YELLOW или GREEN для красного, жёлтого или зелёного сигнала соответственно.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 0 | GREEN |
| 3 | YELLOW |
| 10 | RED |

## 1.4.6. Какой ты, треугольник?

По длинам сторон `a`, `b` и `c` определите вид треугольника: по сторонам — равносторонний, равнобедренный или разносторонний; по углам — прямоугольный, остроугольный или тупоугольный. Если треугольник с такими сторонами построить невозможно, ничего не выводите.

**Формат входных данных**

Три целых числа `a`, `b` и `c` — длины сторон. Ограничение: $1 \le$ `a, b, c` $\le 20000$.

**Формат выходных данных**

Если треугольник существует, выведите в одной строке две характеристики, разделённые одним пробелом. Сначала укажите вид по сторонам:

- `equilateral` — все три стороны равны;
- `isosceles` — равны ровно две стороны;
- `scalene` — все стороны различны.

Затем укажите вид по углам:

- `right triangle` — прямоугольный;
- `oxygon` — остроугольный;
- `obtuse` — тупоугольный.

Если треугольник построить невозможно, вывод должен быть пустым.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3 4 5 | scalene right triangle |
| 2 2 1 | isosceles oxygon |
| 2 2 3 | isosceles obtuse |

Пример 4
Входные данные:

```
1 2 3
```

Выходные данные:

Вывод отсутствует (ни символов, ни перевода строки).

## Блок 1.5. Циклы и обработка последовательностей

### 1.5.1. Найди число-1

Дано число `n`. Найдите минимальное целое число, строго большее `n`, которое делится на 19 без остатка.

**Формат входных данных**

Одно целое число `n`. Ограничение: $1 \le n \le 2^{30}$.

**Формат выходных данных**

Одно целое число — искомое число.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 42 | 57 |
| 19 | 38 |

### 1.5.2. Степень пятёрки

Дано натуральное число `n`. Определите, является ли оно степенью числа 5 с целым неотрицательным показателем. Число 1 считается степенью пятёрки, поскольку $5^0 = 1$.

**Формат входных данных**

Одно целое число `n`. Ограничение: $1 \le n \le 2^{30}$.

**Формат выходных данных**

Выведите `YES`, если `n` является степенью пятёрки, и `NO` в противном случае.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 625 | YES |
| 42 | NO |
| 1 | YES |

### 1.5.3. Средний рост

В классе учатся `n` учеников. Известен рост каждого из них в сантиметрах. Рост мальчиков записан отрицательными числами, рост девочек — положительными; сам рост равен модулю введённого числа. Определите отдельно средний рост мальчиков и средний рост девочек.

**Формат входных данных**

Сначала вводится целое число `n` — количество учеников, затем `n` целых чисел — записи их роста. Ограничения: $2 \le n \le 40$, модуль каждой записи составляет от 30 до 300 включительно. Гарантируется, что в классе есть хотя бы один мальчик и хотя бы одна девочка.

**Формат выходных данных**

Два положительных целых числа в одной строке через пробел: средний рост мальчиков, затем средний рост девочек. Каждый результат округлите вниз до целого числа сантиметров.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 4<br>-170 160 -181 165 | 175 162 |
| 2<br>-180 150 | 180 150 |

## 1.5.4. Максимум последовательности

Дана непустая последовательность целых чисел, заканчивающаяся нулём. Найдите наибольшее число в этой последовательности. Завершающий ноль не является элементом последовательности и при поиске максимума не учитывается.

**Формат входных данных**

Последовательность ненулевых целых чисел, после последнего из которых вводится 0. Последовательность содержит хотя бы одно число и не более $2^{15}$ чисел. Модуль каждого элемента не превышает $2^{15} - 1$.

**Формат выходных данных**

Одно целое число — наибольший элемент последовательности.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1<br>7<br>9<br>0 | 9 |
| -8<br>-2<br>-5<br>0 | -2 |
| -7<br>0 | -7 |

# Блок 1.6. Повторение действий и обработка цифр

## 1.6.1. Hello, World! Новый уровень

Дано натуральное число N. Выведите сообщение `Hello, World!` ровно N раз, каждое с новой строки.

**Формат входных данных**

Одно натуральное число N. Ограничение: $1 \le N < 1000$.

**Формат выходных данных**

Ровно N строк, каждая из которых содержит сообщение `Hello, World!`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 | Hello, World! |
| 3 | Hello, World!<br>Hello, World!<br>Hello, World! |

## 1.6.2. Сумма квадратов до N

По данному натуральному числу n вычислите сумму $1^2 + 2^2 + \ldots + n^2$.

**Формат входных данных**

Одно целое число n. Ограничение: $1 \le n \le 100$.

**Формат выходных данных**

Одно целое число — вычисленная сумма квадратов.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 2 | 5 |
| 100 | 338350 |

## 1.6.3. Все фуд-поинты

На каждом чётном этаже офисного здания расположен один фуд-поинт — комната, где можно выпить чай с печеньем. Офис фирмы занимает этажи с k по n включительно. Выведите номера этажей офиса, на которых есть фуд-поинты, а затем их общее количество.

**Формат входных данных**

Два целых числа: k — номер первого этажа офиса, n — номер последнего этажа. Номера этажей положительные и помещаются в 32-битный знаковый `int`: $1 \le k \le n \le 2^{31} - 1$. Дополнительно $n - k \le 10000$, чтобы объём вывода оставался ограниченным.

**Формат выходных данных**

Выведите номера всех чётных этажей от k до n включительно в порядке возрастания, каждый с новой строки. После них выведите количество этих этажей на отдельной строке. Если чётных этажей в указанном диапазоне нет, выведите только 0.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 6 | 2<br>4<br>6<br>3 |
| 5 5 | 0 |
| 2147483646 2147483647 | 2147483646<br>1 |

## 1.6.4. Измени число

Дано неотрицательное целое число N. Переставьте его цифры так, чтобы сначала шли все чётные цифры, а затем все нечётные. Внутри каждой группы сохраните порядок следования цифр в исходном числе слева направо. Цифра 0 считается чётной. Для хранения полученного числа и промежуточных результатов его построения используйте тип long long: результат может превышать границу 32-битного int, но помещается в знаковый 64-битный тип ($-2^{63}$ … $2^{63} - 1$).

**Формат входных данных**

Одно целое число N. Ограничение: $0 \le N \le 2^{31} - 1$.

**Формат выходных данных**

Одно целое число, полученное перестановкой цифр. Ведущие нули не выводите. Если результат равен нулю, выведите 0.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 12345 | 24135 |
| 1003 | 13 |
| 0 | 0 |

# Блок 1.7. Вложенные циклы

## 1.7.1. N рядов из собачек

Даны N целых чисел $A_i$. Для каждого числа выведите отдельную строку из $A_i$ символов @.

**Формат входных данных**

В первой строке вводится натуральное число N. Во второй строке вводятся N целых чисел $A_i$, разделённых пробелами. Ограничения: $1 \le N \le 1000$, $1 \le A_i \le 20$.

**Формат выходных данных**

Выведите N строк в порядке следования входных чисел. Строка с номером i должна содержать ровно $A_i$ символов @ без пробелов.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 4<br>2 4 1 3 | @@<br>@@@@<br>@<br>@@@ |
| 1<br>1 | @ |

## 1.7.2. Количество делителей — 2

Для каждого целого числа от a до b включительно определите количество его натуральных делителей.

**Формат входных данных**

Два натуральных числа a и b в одной строке. Ограничение: $1 \le a \le b \le 1000$.

**Формат выходных данных**

В одной строке через пробел выведите $b - a + 1$ целых чисел — количества натуральных делителей чисел a, $a + 1$, ..., b в указанном порядке.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 5 | 1 2 2 3 2 |
| 6 6 | 4 |

## 1.7.3. Купюры — 2

Определите количество способов выплатить n рублей монетами номиналом 1, 2, 5 и 10 рублей. Количество монет каждого номинала не ограничено. Способы считаются различными, если отличаются количеством монет хотя бы одного номинала; порядок монет не учитывается.

**Формат входных данных**

Одно натуральное число n. Ограничение: $1 \le n \le 99$.

**Формат выходных данных**

Одно целое число — количество способов выплатить указанную сумму.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5 | 4 |
| 13 | 16 |
| 42 | 220 |

# Блок 1.8. Функции

## 1.8.1. Ограничение команды

Контроллер получает целочисленную команду `value`, которая должна находиться в диапазоне от `lower` до `upper` включительно. Если команда меньше нижней границы, её необходимо заменить нижней границей; если больше верхней — верхней границей. Значение внутри диапазона остаётся прежним.

Реализуйте функцию `int clamp_value(int value, int lower, int upper)`, которая возвращает допустимое значение команды. В функции `main` прочитайте входные данные, вызовите `clamp_value` и выведите её результат. Проверку границ выполняйте внутри `clamp_value`.

**Формат входных данных**

Три целых числа `value`, `lower` и `upper` в одной строке. Ограничения: $-10000 \leq$ `value, lower, upper` $\leq 10000$, `lower` $\leq$ `upper`.

**Формат выходных данных**

Одно целое число — значение команды после ограничения.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| `-5 0 100` | `0` |
| `120 0 100` | `100` |
| `50 0 100` | `50` |

## 1.8.2. Возраст после дня рождения

После каждого дня рождения возраст человека увеличивается на один год. Реализуйте функцию `int next_age(int age)`, которая увеличивает свой параметр на единицу и возвращает полученное значение.

В функции `main` прочитайте исходный возраст, передайте его в `next_age` и сохраните результат в отдельную переменную. Затем выведите исходный возраст и результат вызова. Не присваивайте результат переменной с исходным возрастом: она должна сохранить значение, введённое пользователем.

**Формат входных данных**

Одно целое число `age` — исходный возраст. Ограничение: `0` $\leq$ `age` $\leq$ `120`.

**Формат выходных данных**

Два целых числа в одной строке через пробел: исходный возраст и возраст после следующего дня рождения.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| `18` | `18 19` |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 0 | 0 1 |

### 1.8.3. Простые числа в диапазоне

Найдите количество простых чисел в диапазоне от `a` до `b` включительно. Простое число — натуральное число больше 1, у которого ровно два натуральных делителя: 1 и само число.

Реализуйте две функции:

- `bool is_prime(int number)` — возвращает `true`, если число простое, и `false` в противном случае;
- `int count_primes(int a, int b)` — возвращает количество простых чисел в диапазоне, вызывая `is_prime` для проверки каждого числа.

В функции `main` прочитайте границы диапазона, вызовите `count_primes` и выведите результат. Функции `is_prime` и `count_primes` не должны выполнять ввод или вывод данных.

**Формат входных данных**

Два целых числа `a` и `b` в одной строке. Ограничение: $1 \leq a \leq b \leq 1000$.

**Формат выходных данных**

Одно целое число — количество простых чисел в указанном диапазоне.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 10 | 4 |
| 14 16 | 0 |
| 2 2 | 1 |

# Глава 2

## Блок 2.1. Массивы

### 2.1.1. Кратность k

Дан массив из n целых чисел. Выведите все элементы массива, кратные числу k, сохранив порядок их следования в исходном массиве.

**Формат входных данных**

В первой строке вводится целое число n — количество элементов массива. Во второй строке вводятся n целых чисел $a_i$, разделённых пробелами. В третьей строке вводится целое число k. Ограничения: $1 \le n \le 100$, $0 \le a_i < 1000000$, $1 \le k < 1000$.

**Формат выходных данных**

Выведите элементы массива, кратные k, в одной строке через пробел. Если таких элементов нет, выведите пустую строку.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>1 2 3 4 5<br>4 | 4 |
| 4<br>0 6 7 12<br>3 | 0 6 12 |
| 3<br>1 5 7<br>2 | Пустая строка (один перевод строки). |

### 2.1.2. Минимум в массиве

Дан массив из N целых чисел. Напишите программу, которая найдёт и выведет минимальное значение в этом массиве.

**Формат входных данных**

В первой строке вводится целое число N — количество элементов массива. Во второй строке вводятся N целых чисел, разделённых пробелами. Ограничения: $1 \le N \le 100$, $-10^9 \le a_i \le 10^9$.

**Формат выходных данных**

Одно целое число — минимальное значение в массиве.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>4 3 6 1 2 | 1 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>-5 -2 -8 | -8 |

## 2.1.3. Сдвиг вправо

Дан массив из n целых чисел и два индекса s и k, причём s > k. Перенесите элемент с индексом s на позицию с индексом k. Элементы, первоначально находившиеся на позициях от k до s − 1 включительно, сдвиньте вправо на одну позицию. Остальные элементы должны остаться на своих местах. Индексация начинается с нуля.

**Формат входных данных**

В первой строке вводится целое число n — количество элементов массива. Во второй строке вводятся n целых чисел, разделённых пробелами. В третьей строке вводятся два целых числа s и k именно в этом порядке. Ограничения: $2 \le n \le 100$, $-100 \le a_i \le 100$, $0 \le k < s < n$.

**Формат выходных данных**

Выведите изменённый массив из n элементов в одной строке через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>4 5 3 1 2<br>3 1 | 4 1 5 3 2 |
| 2<br>7 9<br>1 0 | 9 7 |
| 5<br>1 2 3 4 5<br>4 0 | 5 1 2 3 4 |

## 2.1.4. Удалить элемент

Дан массив из n целых чисел и индекс k. Удалите элемент с индексом k, сдвинув влево на одну позицию все элементы, расположенные правее него. Порядок оставшихся элементов должен сохраниться. Индексация начинается с нуля.

**Формат входных данных**

В первой строке вводится целое число n — количество элементов массива. Во второй строке вводятся n целых чисел, разделённых пробелами. В третьей строке вводится целое число k — индекс удаляемого элемента. Ограничения: $1 \le n \le 15$, $-1000 \le a_i \le 1000$, $0 \le k < n$.

**Формат выходных данных**

Выведите оставшиеся n − 1 элементов в одной строке через пробел. Если массив стал пустым, выведите пустую строку.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 7<br>7 6 5 4 3 2 1<br>2 | 7 6 4 3 2 1 |
| 1<br>42<br>0 | Пустая строка (один перевод строки). |
| 3<br>1 2 3<br>2 | 1 2 |

## 2.1.5. Вставить элемент — 1

Дан массив из `n` целых чисел. Вставьте число `x` на позицию с индексом `k`, сдвинув элементы с индексами от `k` до `n − 1` вправо на одну позицию. При `k = n` добавьте число в конец массива. Индексация начинается с нуля. После вставки массив должен содержать `n + 1` элементов; предусмотрите место для дополнительного элемента.

**Формат входных данных**

В первой строке вводится целое число `n` — количество элементов исходного массива. Во второй строке вводятся `n` целых чисел, разделённых пробелами. В третьей строке вводятся два целых числа `k` и `x` именно в этом порядке. Ограничения: $1 \le n \le 1000$, $-10^9 \le a_i, x \le 10^9$, $0 \le k \le n$.

**Формат выходных данных**

Выведите массив после вставки: `n + 1` целых чисел в одной строке через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>2 7 1<br>0 100 | 100 2 7 1 |
| 3<br>2 7 1<br>1 100 | 2 100 7 1 |
| 3<br>2 7 1<br>3 100 | 2 7 1 100 |

## 2.1.6. Удаление повторов

Дан массив из `n` целых чисел. Удалите повторные вхождения каждого значения, оставив только его первое появление. Порядок оставшихся элементов должен совпадать с их порядком в исходном массиве.

**Формат входных данных**

В первой строке вводится целое число `n` — количество элементов массива. Во второй строке вводятся `n` целых чисел, разделённых пробелами. Ограничения: $1 \le n \le 10000$, $1 \le a_i \le 1000000$.

**Формат выходных данных**

Выведите массив после удаления повторов в одной строке через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 8<br>5 13 6 3 13 4 5 9 | 5 13 6 3 4 9 |
| 4<br>7 7 7 7 | 7 |
| 3<br>3 1 2 | 3 1 2 |

## 2.1.7. Что ты нашёл?

Боря нашёл улику и хочет сообщить об этом Аркадию. Прочитайте слово, обозначающее находку, и выведите фразу `Аркадий, смотри, я нашёл WORD !`, заменив `WORD` введённым словом. Сохраните написание слова без изменений. Перед восклицательным знаком должен быть один пробел, как в образце фразы.

Храните слово в массиве `char word[101]`. Строка C — это массив символов, заканчивающийся нулевым символом `'\0'`: для слова из 100 байт нужен массив как минимум из 101 элемента. В UTF-8 русская буква может занимать несколько байт, поэтому ограничение здесь задано в байтах, а не в количестве букв.

В C подключите `<stdio.h>` и считайте слово с помощью `scanf("%100s word)`, проверив, что возвращено значение 1. Число 100 ограничивает количество считываемых байт; завершающий ноль записывается дополнительно. Перед `word` в этом вызове не ставится `&`: имя массива передаёт адрес его первого элемента. Для вывода слова используйте формат `%s` внутри строки формата `printf`.

Решение можно написать и на C++, используя тот же массив и функции ввода/вывода. Для этой задачи использование `std::string` не допускается: требуется попрактиковаться именно с массивом символов. Исходный файл с русской фразой и входные/выходные данные должны быть в кодировке UTF-8.

**Формат входных данных**

Одно непустое слово без пробельных символов, занимающее не более 100 байт в кодировке UTF-8. Слово состоит из русских или латинских букв.

**Формат выходных данных**

Одна строка: `Аркадий, смотри, я нашёл WORD !`, где вместо `WORD` находится введённое слово. Например, для слова `фантик` выведите `Аркадий, смотри, я нашёл фантик !`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| фантик | Аркадий, смотри, я нашёл фантик ! |
| apple | Аркадий, смотри, я нашёл apple ! |

## 2.1.8. Заполнение двумерного массива

Заполните двумерный массив из N строк и N столбцов. Первая строка содержит числа от 1 до N в порядке возрастания. Каждая следующая строка получается циклическим сдвигом предыдущей строки влево на одну позицию: первый элемент переносится в конец строки, остальные сдвигаются влево.

**Формат входных данных**

Одно целое число N. Ограничение: $1 \le N \le 100$.

**Формат выходных данных**

Выведите заполненный массив: N строк, в каждой из которых находятся N целых чисел, разделённых пробелами.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3 | 1 2 3<br>2 3 1<br>3 1 2 |
| 1 | 1 |

## 2.1.9. Элементы над главной диагональю

Дан двумерный массив размера $N \times N$, заполненный двузначными целыми числами. Выведите все элементы, расположенные строго выше главной диагонали. При индексации с нуля это элементы с индексами строки i и столбца j, для которых $j > i$. Обходите строки сверху вниз, а элементы внутри каждой строки — слева направо.

**Формат входных данных**

В первой строке вводится целое число N — количество строк и столбцов массива. Следующие N строк содержат по N целых чисел, разделённых пробелами. Ограничения: $1 \le N \le 500$, $10 \le |a_{ij}| \le 99$.

**Формат выходных данных**

Выведите элементы, расположенные строго выше главной диагонали, в одной строке через пробел в указанном порядке обхода. При $N = 1$ выведите пустую строку.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>12 27 15<br>44 79 13<br>62 48 99 | 27 15 13 |
| 1<br>11 | Пустая строка (один перевод строки). |
| 2<br>-10 -20<br>30 40 | -20 |

## 2.1.10. Пузырьковая сортировка

Дан массив из `n` целых чисел. Отсортируйте его по неубыванию методом пузырьковой сортировки: последовательно сравнивайте соседние элементы и меняйте их местами, если они расположены в неправильном порядке. Повторяйте проходы, пока массив не будет отсортирован.

Реализуйте сортировку в отдельной функции `void bubble_sort(int array[], int n)`. Функция должна изменять переданный массив и не выполнять ввод или вывод. В функции `main` прочитайте массив, вызовите `bubble_sort` и выведите результат. Использование готовых функций сортировки, включая `qsort` и `std::sort`, запрещено.

### Материал к заданию

Основные виды сортировок и примеры их реализации — Яндекс Образование, раздел «Пузырьковая сортировка и её улучшения». Реализацию из статьи адаптируйте к массиву и интерфейсу `bubble_sort`, заданным в условии.

Сложность алгоритмов. Разбор Big O — Хабр — материал об асимптотической оценке сложности алгоритмов.

### Формат входных данных

В первой строке вводится целое число `n` — количество элементов массива. Во второй строке вводятся `n` целых чисел, разделённых пробелами. Ограничения: $1 \le n \le 1000$, $-10^9 \le a_i \le 10^9$. Значения могут повторяться; массив может быть уже отсортирован или расположен в обратном порядке.

### Формат выходных данных

Выведите `n` элементов отсортированного массива в одной строке через пробел. Каждый следующий элемент должен быть не меньше предыдущего. Все исходные элементы, включая повторяющиеся значения, должны сохраниться.

### Примеры

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>3 -1 3 0 2 | -1 0 2 3 3 |
| 4<br>1 2 3 4 | 1 2 3 4 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 4<br>4 3 2 1 | 1 2 3 4 |

## 2.1.11. Двоичный поиск

Дан массив из `n` целых чисел, отсортированный по неубыванию, и целое число `x`. Найдите индекс первого вхождения числа `x` в массив. Если число отсутствует, результатом поиска должно быть $-1$. Индексация начинается с нуля.

Реализуйте поиск в отдельной функции `int binary_search_first(const int array[], int n, int x)`. Используйте алгоритм двоичного поиска: сравнивайте искомое число с элементом в середине рассматриваемого диапазона и сокращайте диапазон поиска. Если найден элемент, равный `x`, продолжайте поиск первого вхождения среди элементов левее него. Функция не должна изменять массив или выполнять ввод и вывод.

В функции `main` прочитайте входные данные, вызовите `binary_search_first` и выведите её результат. Линейный перебор массива и использование готовых функций поиска, включая `bsearch`, `std::binary_search` и `std::lower_bound`, запрещены. Количество проверяемых элементов должно расти не быстрее `O(log n)`; повторяющиеся значения не должны приводить к линейному поиску первого вхождения.

### Материал к заданию

Бинарный поиск — Хабр. Примеры написаны на JavaScript. Перенесите принцип сокращения диапазона поиска на C или C++, соблюдая интерфейс из условия. Для этой задачи требуется первое вхождение: после обнаружения равного элемента продолжайте поиск слева; линейный проход по повторяющимся элементам не допускается.

### Формат входных данных

В первой строке вводится целое число `n` — количество элементов массива. Во второй строке вводятся `n` целых чисел, разделённых пробелами, в порядке неубывания. В третьей строке вводится целое число `x`. Ограничения: $1 \le n \le 10000$, $-10^9 \le a_i, x \le 10^9$. Значения элементов могут повторяться.

### Формат выходных данных

Одно целое число — индекс первого вхождения `x` в массив либо $-1$, если такого элемента нет.

### Примеры

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 6<br>-3 0 2 2 2 8<br>2 | 2 |
| 4<br>1 3 5 7<br>4 | -1 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1<br>9<br>9 | 0 |

## 2.1.12. Быстрая сортировка

Дан массив из `n` целых чисел. Отсортируйте его по неубыванию алгоритмом быстрой сортировки (Quick Sort). Выберите опорное значение, разделите обрабатываемую часть массива относительно него и отсортируйте полученные части тем же алгоритмом. Части, содержащие менее двух элементов, не требуют дальнейшей обработки.

Реализуйте алгоритм в отдельной функции `void quick_sort(int array[], int left, int right)`, где `left` и `right` — включительные границы обрабатываемой части массива, индексация начинается с нуля. Функция должна изменять переданный массив и не выполнять ввод или вывод. В функции `main` прочитайте массив, вызовите `quick_sort(array, 0, n - 1)` и выведите результат. Для обработки частей используйте рекурсивные вызовы: функция вызывает сама себя для меньших частей массива.

Использование готовых функций сортировки, включая `qsort` и `std::sort`, запрещено. Алгоритм должен корректно обрабатывать повторяющиеся значения, в том числе массив из одинаковых чисел. При разделении границы должны продвигаться даже при равенстве элементов опорному значению, а последующие вызовы должны получать меньшие части. Учитывайте, что выбор опорного значения влияет на скорость сортировки и глубину рекурсии; проверьте решение также на уже отсортированном массиве и массиве в обратном порядке.

### Материал к заданию

Основные виды сортировок и примеры их реализации — Яндекс Образование, раздел «Быстрая сортировка». Реализацию из статьи адаптируйте к массиву и включительным границам `left`, `right`, заданным в условии. Выбор последнего элемента в качестве опорного, использованный в статье, может приводить к квадратичному числу операций и линейной глубине рекурсии на отсортированных массивах и массивах из одинаковых значений; при разработке учитывайте указанные в условии случаи.

### Формат входных данных

В первой строке вводится целое число `n` — количество элементов массива. Во второй строке вводятся `n` целых чисел, разделённых пробелами. Ограничения: $1 \le n \le 10000$, $-10^9 \le a_i \le 10^9$. Значения могут повторяться.

### Формат выходных данных

Выведите `n` элементов отсортированного массива в одной строке через пробел. Каждый следующий элемент должен быть не меньше предыдущего. Все исходные элементы, включая повторяющиеся значения, должны сохраниться.

### Примеры

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 6<br>5 -2 0 5 1 -2 | -2 -2 0 1 5 5 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 4<br>7 7 7 7 | 7 7 7 7 |
| 5<br>5 4 3 2 1 | 1 2 3 4 5 |

## Блок 2.2. Указатели и ссылки

### 2.2.1. Обмен значений

Даны два целых числа `a` и `b`. Реализуйте функцию `void swap_values(int *a, int *b)`, которая меняет местами значения двух переменных, переданных ей через указатели. Обмен выполняйте внутри функции, изменяя объекты по переданным адресам. Функция не должна выполнять ввод или вывод. Переданные указатели указывают на существующие переменные типа `int` и не равны нулевому указателю.

В функции `main` прочитайте два числа в отдельные переменные, вызовите `swap_values(&a, &b)` и выведите значения переменных после вызова. Использование готовых функций обмена, включая `std::swap`, запрещено. Решение через указатели можно написать как на C, так и на C++.

Опционально, для знакомых со ссылками в C++: вместо варианта с указателями реализуйте функцию `void swap_values(int &a, int &b)` и вызовите её как `swap_values(a, b)`. Значения исходных переменных также должны изменяться внутри функции. Этот вариант доступен только в C++ и не является обязательным.

**Формат входных данных**

Два целых числа `a` и `b` в одной строке через пробел. Оба числа находятся в диапазоне 32-битного знакового `int`: $-2^{31} \leq$ `a, b` $\leq 2^{31} - 1$.

**Формат выходных данных**

Два целых числа в одной строке через пробел — значения переменных `a` и `b` после обмена. Первым выведите исходное значение `b`, вторым — исходное значение `a`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5 -2 | -2 5 |
| 7 7 | 7 7 |
| -2147483648 2147483647 | 2147483647 -2147483648 |

### 2.2.2. Добавить в конец

Дан массив из `N` целых чисел и целое число `value`. Реализуйте функцию `void push_back(int array[], int *size, int value)`, которая добавляет `value` в конец массива и увеличивает его логический размер на единицу. Текущий размер передаётся через указатель `size`; его изменение внутри функции должно сохраняться после вызова. В массиве заранее предусмотрено место как минимум для `N + 1` элементов. Изменять размер выделенной памяти не требуется.

В функции `main` прочитайте массив и добавляемое значение, вызовите `push_back(array, &size, value)` и выведите массив, используя обновлённый размер. Функция `push_back` не должна выполнять ввод или вывод. Использование готовых контейнеров и их операций добавления элементов запрещено.

Опционально, для знакомых со ссылками в C++: реализуйте вариант `void push_back(int array[], int &size, int value)` и вызовите его как `push_back(array, size, value)`.

**Формат входных данных**

В первой строке вводится целое число `N` — количество элементов исходного массива. Во второй строке вводятся `N` целых чисел, разделённых пробелами. В третьей строке вводится целое число `value`. Ограничения: $1 \le N \le 100$, $-10^9 \le a_i, value \le 10^9$.

**Формат выходных данных**

Выведите массив после добавления элемента: `N + 1` целых чисел в одной строке через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>1 2 3<br>4 | 1 2 3 4 |
| 1<br>-5<br>0 | -5 0 |

## 2.2.3. Функция минимума

Дан массив из `N` целых чисел. Реализуйте функцию `int min_index(const int data[], int size)`, которая возвращает индекс первого минимального элемента массива. Если минимальное значение встречается несколько раз, верните наименьший из его индексов. Индексация начинается с нуля.

Функция не должна изменять массив или выполнять ввод и вывод. В функции `main` прочитайте массив, вызовите `min_index` и выведите возвращённый индекс. При вызове функции `size` положителен и соответствует количеству доступных элементов массива.

**Формат входных данных**

В первой строке вводится целое число `N` — количество элементов массива. Во второй строке вводятся `N` целых чисел, разделённых пробелами. Ограничения: $1 \le N \le 100$, $-10^6 \le a_i \le 10^6$.

**Формат выходных данных**

Одно целое число — индекс первого минимального элемента массива.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>2 10 -1 7 -7 | 4 |
| 4<br>3 -2 -2 5 | 1 |
| 1<br>42 | 0 |

### 2.2.4. Функция «Переворот массива»

Дан массив из N целых чисел. Реализуйте функцию `void reverse_array(int array[], int size)`, которая располагает элементы переданного массива в обратном порядке. Изменяйте исходный массив, обменивая его элементы; создавать второй массив для хранения результата запрещено. Размер массива после переворота не меняется.

Функция не должна выполнять ввод или вывод. В функции `main` прочитайте массив, вызовите `reverse_array` и выведите изменённый массив. Для обмена элементов можно использовать функцию `swap_values` из задачи 2.2.1. Использование готовых функций переворота, включая `std::reverse`, запрещено.

Дополнительно, по желанию, выделите ввод и вывод в отдельные функции `void read_array(int array[], int size)` и `void print_array(const int array[], int size)`. Количество элементов прочитайте в `main` до вызова `read_array`.

**Формат входных данных**

В первой строке вводится целое число N — количество элементов массива. Во второй строке вводятся N целых чисел, разделённых пробелами. Ограничения: $1 \le N \le 100$, $-2^{30} \le a_i \le 2^{30}$.

**Формат выходных данных**

Выведите N элементов массива в обратном порядке в одной строке через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>4 6 9 2 1 | 1 2 9 6 4 |
| 4<br>1 2 3 4 | 4 3 2 1 |
| 1<br>7 | 7 |

### 2.2.5. Двойной переворот

Создайте массив, содержащий числа от 1 до N в порядке возрастания. Сначала расположите в обратном порядке элементы на позициях от A до B включительно, затем — элементы на позициях от C до D включительно. Второй переворот выполняется в массиве, уже изменённом первым переворотом. Участки могут пересекаться. В условии позиции нумеруются с единицы.

Реализуйте функцию `void reverse_range(int array[], int left, int right)`, которая переворачивает участок исходного массива между включительными границами `left` и `right`. Параметры функции являются индексами с нуля. В функции `main` вызовите её дважды: сначала с границами `A - 1` и `B - 1`, затем с границами `C - 1` и `D - 1`.

Функция должна изменять переданный массив, не выполнять ввод или вывод и не создавать второй массив для хранения результата. Для обмена элементов можно использовать функцию `swap_values` из задачи 2.2.1. Использование готовых функций переворота, включая `std::reverse`, запрещено.

**Формат входных данных**

Пять целых чисел `N`, `A`, `B`, `C`, `D` в одной строке через пробел. Ограничения: $2 \le N \le 1000$, $1 \le A < B \le N$, $1 \le C < D \le N$.

**Формат выходных данных**

Выведите полученный массив из `N` целых чисел в одной строке через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 9 2 5 6 9 | 1 5 4 3 2 9 8 7 6 |
| 9 3 6 5 8 | 1 2 6 5 8 7 3 4 9 |
| 4 1 4 1 4 | 1 2 3 4 |

## Блок 2.3. Структуры

### 2.3.1. Ближайшая точка

Даны координаты `N` точек на плоскости. Найдите точку, ближайшую к началу координат, и выведите её номер. Если несколько точек находятся на одинаковом минимальном расстоянии, выберите первую во входных данных. Номера точек начинаются с единицы.

Объявите структуру `struct Point` с целочисленными полями `x` и `y`. Сохраните точки в массиве структур. Реализуйте функцию `int nearest_point(const struct Point points[], int count)`, которая возвращает индекс выбранной точки с нуля, не изменяет массив и не выполняет ввод или вывод. В функции `main` прочитайте данные, вызовите функцию и преобразуйте полученный индекс в номер точки.

Для сравнения расстояний достаточно сравнивать значения $x^2 + y^2$. Используйте для их вычисления тип `long long`, выполняя преобразование до умножения: при заданных ограничениях квадрат координаты может не помещаться в 32-битный `int`.

**Формат входных данных**

В первой строке вводится целое число `N` — количество точек. В следующих `N` строках вводятся по два целых числа $x_i$ и $y_i$ — координаты очередной точки. Ограничения: $1 \le N \le 1000$, $-10^6 \le x_i, y_i \le 10^6$.

**Формат выходных данных**

Одно целое число — номер ближайшей к началу координат точки, от 1 до `N`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>3 4<br>1 1<br>5 0 | 2 |
| 3<br>1 0<br>0 -1<br>2 2 | 1 |
| 2<br>1000000 1000000<br>0 0 | 2 |

## 2.3.2. Прибавить секунды

Дано время суток: часы, минуты и секунды. Определите время, которое наступит через `K` секунд. При переходе через полночь отсчёт времени начинается снова с `0 0 0`. За указанный промежуток может пройти несколько суток; количество прошедших суток выводить не требуется.

Объявите структуру `struct Time` с целочисленными полями `hours`, `minutes` и `seconds`. Реализуйте функцию `void add_seconds(struct Time *time, int delta)`, которая изменяет время в переданной структуре. Указатель указывает на существующую структуру с корректным временем, а `delta` неотрицателен. После вызова поля должны оставаться в допустимых диапазонах. Функция не должна выполнять ввод или вывод.

В функции `main` прочитайте время и `K`, вызовите `add_seconds` и выведите изменённые поля. Опционально, для знакомых со ссылками в C++: используйте сигнатуру `void add_seconds(struct Time &time, int delta)`.

**Формат входных данных**

В первой строке вводятся три целых числа `h`, `m`, `s` — часы, минуты и секунды. Во второй строке вводится целое число `K`. Ограничения: $0 \le h \le 23$, $0 \le m, s \le 59$, $0 \le K \le 10^9$.

**Формат выходных данных**

Три целых числа в одной строке через пробел — часы, минуты и секунды после прибавления `K` секунд. Ведущие нули не требуются.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 12 30 15<br>45 | 12 31 0 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 23 59 59<br>1 | 0 0 0 |
| 1 2 3<br>172800 | 1 2 3 |

### 2.3.3. Лучший участник

В соревновании участвуют N человек. Для каждого известны числовой идентификатор и результаты трёх этапов. Побеждает участник с наибольшей суммой результатов. При равенстве сумм побеждает участник с меньшим идентификатором.

Объявите структуру `struct Participant` с целочисленными полями `id`, `score1`, `score2` и `score3`. Сохраните участников в массиве структур. Реализуйте функцию `int best_participant(const struct Participant participants[], int count)`, которая возвращает индекс победителя с нуля. Функция не должна изменять массив или выполнять ввод и вывод. В функции `main` прочитайте данные, вызовите функцию и выведите идентификатор выбранного участника.

**Формат входных данных**

В первой строке вводится целое число N — количество участников. Следующие N строк содержат по четыре целых числа: `id`, `score1`, `score2`, `score3`. Ограничения: $1 \le$ N $\le 1000$, $1 \le$ `id` $\le 10^9$, $0 \le$ `score1, score2, score3` $\le 100$. Идентификаторы участников различны.

**Формат выходных данных**

Одно целое число — идентификатор победителя.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>10 20 30 40<br>7 50 50 50<br>2 0 0 0 | 7 |
| 2<br>20 100 0 0<br>5 50 25 25 | 5 |
| 1<br>99 0 0 0 | 99 |

### 2.3.4. Статистика датчика

Дан набор из N измерений датчика. Каждое измерение содержит время в секундах от начала наблюдения и температуру в десятых долях градуса Цельсия. Температура считается допустимой, если её значение находится в диапазоне от 0 до 500 включительно. Остальные измерения исключаются из расчётов.

Найдите количество допустимых измерений, минимальную и максимальную температуру и среднюю температуру среди допустимых измерений. Время измерения необ-

ходимо сохранить вместе с температурой, но в этой задаче оно не влияет на расчёты и не выводится.

Объявите структуру `struct Measurement` с целочисленными полями `time_s` и `raw_temperature`. Сохраните измерения в массиве структур. Объявите структуру `struct Statistics` с целочисленными полями `valid_count`, `min_raw`, `max_raw` и полем `average_c` типа `double`.

Реализуйте функцию `struct Statistics analyse_measurements(const struct Measurement measurements[], int count)`, которая возвращает статистику, не изменяет массив и не выполняет ввод или вывод. В полях `min_raw` и `max_raw` храните температуру в десятых долях градуса, в `average_c` — среднюю температуру в градусах Цельсия. Если допустимых измерений нет, все поля результата должны быть равны нулю; признаком отсутствия данных служит `valid_count == 0`.

В функции `main` прочитайте данные, вызовите функцию и выведите результат. При вычислении среднего каждое допустимое измерение учитывается один раз; усреднять по длительности между измерениями не требуется.

**Формат входных данных**

В первой строке вводится целое число `N` — количество измерений. В следующих `N` строках вводятся по два целых числа `time_s` и `raw_temperature`. Ограничения: $1 \le$ `N` $\le 1000$, $0 \le$ `time_s` $\le 10^9$, $-1000 \le$ `raw_temperature` $\le 1000$. Времена измерений заданы в порядке неубывания и могут повторяться.

**Формат выходных данных**

Если допустимых измерений нет, выведите строку `NO DATA`.

Иначе выведите в одной строке через пробел четыре значения: количество допустимых измерений, минимальную температуру, максимальную температуру и среднюю температуру. Все три температуры выведите в градусах Цельсия с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>0 240<br>1 250<br>2 260<br>3 -1<br>4 280 | 4 24.000000 28.000000 25.750000 |
| 2<br>0 -1<br>1 501 | NO DATA |
| 2<br>0 0<br>0 500 | 2 0.000000 50.000000 25.000000 |

## 2.3.5. Сложение дробей

Даны две дроби с целыми числителями и положительными знаменателями. Найдите их сумму и представьте её в виде несократимой дроби с положительным знаменателем.

Если сумма равна нулю, результат должен иметь числитель 0 и знаменатель 1. Выделять целую часть не требуется.

Объявите структуру `struct Fraction` с полями `numerator` и `denominator` типа `long long`. Реализуйте функцию `struct Fraction add_fractions(struct Fraction first, struct Fraction second)`, которая принимает две структуры по значению и возвращает структуру с сокращённой суммой. Функция не должна выполнять ввод или вывод. Для сокращения реализуйте отдельную функцию вычисления наибольшего общего делителя (НОД). При вычислении НОД учитывайте, что числитель может быть отрицательным.

В функции `main` прочитайте дроби, вызовите `add_fractions` и выведите поля возвращённой структуры. Все промежуточные вычисления выполняйте в `long long`: произведения знаменателей и числителей могут не помещаться в 32-битный `int`.

**Материал к заданию**

Алгоритм Евклида — Algorithmica. Используйте разбор вычисления НОД для реализации сокращения суммы дробей; в своей функции учитывайте отрицательный числитель и представление нулевого результата, заданные в условии.

**Формат входных данных**

В первой строке вводятся два целых числа $p_1$ и $q_1$ — числитель и знаменатель первой дроби. Во второй строке вводятся $p_2$ и $q_2$ — числитель и знаменатель второй дроби. Ограничения: $-10^6 \le p_1, p_2 \le 10^6$, $1 \le q_1, q_2 \le 10^6$.

**Формат выходных данных**

Два целых числа в одной строке через пробел — числитель и знаменатель сокращённой суммы.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 1 2<br>1 3 | 5 6 |
| -1 2<br>1 2 | 0 1 |
| -3 4<br>1 2 | -1 4 |

## Блок 2.4. Перечисления

### 2.4.1. Состояние устройства

Устройство может находиться в одном из четырёх состояний. Объявите перечисление `enum DeviceState` с явно заданными значениями: `DEVICE_OFF = 0` — выключено, `DEVICE_READY = 1` — готово, `DEVICE_MEASURING = 2` — выполняет измерение, `DEVICE_ERROR = 3` — ошибка.

Прочитайте числовой код и выведите название соответствующего состояния. Для выбора используйте `switch` с константами перечисления в метках `case`. Если код не соответствует ни одному состоянию, выведите `UNKNOWN`. Само наличие перечисления не заменяет проверку введённого кода.

**Формат входных данных**

Одно целое число `code`. Ограничение: $-100 \leq$ `code` $\leq 100$.

**Формат выходных данных**

Одна строка: `OFF`, `READY`, `MEASURING` или `ERROR` для кодов 0, 1, 2 или 3 соответственно. Для любого другого кода выведите `UNKNOWN`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 0 | OFF |
| 2 | MEASURING |
| 9 | UNKNOWN |

## 2.4.2. Управление устройством

Даны начальное состояние устройства и последовательность команд. Определите состояние после выполнения всех команд в заданном порядке.

Объявите `enum DeviceState` с константами `DEVICE_OFF = 0`, `DEVICE_READY = 1`, `DEVICE_MEASURING = 2`, `DEVICE_ERROR = 3`. Объявите `enum Command` с константами `CMD_POWER_ON = 1`, `CMD_START = 2`, `CMD_STOP = 3`, `CMD_FAIL = 4`, `CMD_RESET = 5`, `CMD_POWER_OFF = 6`.

Правила выполнения команд:

- `CMD_POWER_ON`: выключенное устройство переходит в состояние готовности; остальные состояния не изменяются.
- `CMD_START`: готовое устройство начинает измерение; остальные состояния не изменяются.
- `CMD_STOP`: измеряющее устройство переходит в состояние готовности; остальные состояния не изменяются.
- `CMD_FAIL`: любое включённое устройство переходит в состояние ошибки; выключенное остаётся выключенным.
- `CMD_RESET`: устройство в состоянии ошибки переходит в состояние готовности; остальные состояния не изменяются.
- `CMD_POWER_OFF`: устройство из любого состояния переходит в выключенное состояние.

Реализуйте функцию `enum DeviceState apply_command(enum DeviceState state, enum Command command)`, которая принимает состояние и команду по значению и возвращает новое состояние. Функция не должна выполнять ввод или вывод. В `main` вызовите её для каждой команды, каждый раз сохраняя возвращённое состояние.

Входные коды гарантированно допустимы. В C++ для преобразования прочитанного целого к соответствующему перечислению используйте явное приведение типа, например `static_cast<enum Command>(code)`.

**Формат входных данных**

В первой строке вводится целое число `state` — начальное состояние. Во второй строке вводится целое число `N` — количество команд. В третьей строке вводятся `N` целых кодов команд через пробел. Ограничения: $0 \le$ `state` $\le 3$, $1 \le$ `N` $\le 1000$, каждый код команды находится в диапазоне от 1 до 6.

**Формат выходных данных**

Одна строка — название конечного состояния: `OFF`, `READY`, `MEASURING` или `ERROR`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 0<br>2<br>1 2 | MEASURING |
| 1<br>4<br>2 4 2 5 | READY |
| 0<br>3<br>2 4 5 | OFF |

## Блок 2.5. Побитовые операции

### 2.5.1. Проверить бит

Даны беззнаковое целое число `value` и номер бита `k`. Определите значение этого бита. Биты нумеруются с нуля от младшего к старшему: бит 0 соответствует значению 1, бит 1 — значению 2 и так далее.

Используйте тип `unsigned int` и побитовые операции. Для этих задач предполагается, что `unsigned int` имеет размер 32 бита. Маски формируйте из беззнаковых значений, например `1u`, чтобы операции сдвига выполнялись в беззнаковом типе. Перевод числа в строку двоичных цифр не допускается.

**Формат входных данных**

Два числа в одной строке: `value` и `k`. Ограничения: $0 \le$ `value` $\le 2^{32} - 1$, $0 \le$ `k` $\le 31$.

**Формат выходных данных**

Одно целое число: 0, если указанный бит сброшен, или 1, если он установлен.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 10 1 | 1 |
| 10 0 | 0 |
| 2147483648 31 | 1 |

### 2.5.2. Изменить бит

Даны беззнаковое целое число `value`, номер бита `k` и код команды `command`. Выполните одну операцию: при команде 1 установите бит в 1, при команде 2 сбросьте его в 0, при команде 3 переключите его значение на противоположное. Все остальные биты должны сохраниться.

Используйте 32-битный `unsigned int` и побитовые операции. Маски формируйте в беззнаковом типе. Перевод числа в строку двоичных цифр не допускается.

**Формат входных данных**

Три числа в одной строке: `value`, `k`, `command`. Ограничения: $0 \le$ `value` $\le 2^{32} - 1$, $0 \le$ `k` $\le 31$, $1 \le$ `command` $\le 3$.

**Формат выходных данных**

Одно беззнаковое целое число в десятичной записи — результат изменения указанного бита.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 8 0 1 | 9 |
| 15 2 2 | 11 |
| 0 31 3 | 2147483648 |

### 2.5.3. Проверить флаги

Каждый бит числа `flags` представляет независимый флаг: 1 означает, что флаг установлен, 0 — что сброшен. В числе `mask` единицами отмечены флаги, которые необходимо проверить.

Определите отдельно, установлен ли хотя бы один из отмеченных флагов и установлены ли все отмеченные флаги. Нулевая маска во входных данных не встречается. Используйте 32-битный `unsigned int` и побитовые операции. Перевод чисел в строки двоичных цифр не допускается.

**Формат входных данных**

Два числа в одной строке: `flags` и `mask`. Ограничения: $0 \le$ `flags` $\le 2^{32} - 1$, $1 \le$ `mask` $\le 2^{32} - 1$.

**Формат выходных данных**

Выведите две строки. В первой — `YES`, если установлен хотя бы один флаг из маски, иначе `NO`. Во второй — `YES`, если установлены все флаги из маски, иначе `NO`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 10 2 | YES<br>YES |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 10 6 | YES<br>NO |
| 8 3 | NO<br>NO |

### 2.5.4. Упаковать настройки

Настройки устройства хранятся в младших 16 битах беззнакового числа `settings`. Биты 0–2 включительно содержат режим, биты 3–7 включительно — мощность. Биты 8–15 содержат другие настройки, которые необходимо сохранить без изменений. Биты нумеруются с нуля от младшего к старшему.

Даны исходное число и новые значения режима и мощности. Замените оба поля, сохранив остальные биты. Затем извлеките режим и мощность из полученного числа. Используйте тип `unsigned int`, побитовые маски и сдвиги; перевод числа в строку двоичных цифр не допускается.

Реализуйте две функции: `unsigned int update_settings(unsigned int settings, unsigned int mode, unsigned int power)` — возвращает число с обновлёнными полями; `void unpack_settings(unsigned int settings, unsigned int *mode, unsigned int *power)` — записывает извлечённые значения через указатели на существующие переменные. Функции не должны выполнять ввод или вывод. В `main` сначала вызовите `update_settings`, затем передайте её результат в `unpack_settings` и выведите результаты.

**Формат входных данных**

Три числа в одной строке: `settings`, `mode`, `power`. Ограничения: $0 \le$ `settings` $\le 65535$, $0 \le$ `mode` $\le 7$, $0 \le$ `power` $\le 31$.

**Формат выходных данных**

Три беззнаковых целых числа в одной строке через пробел: обновлённое число настроек, режим и мощность, извлечённые из этого числа. Все числа выводятся в десятичной записи.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 0 3 10 | 83 3 10 |
| 65535 0 0 | 65280 0 0 |
| 256 7 31 | 511 7 31 |

# Глава 3

## Блок 3.1. Динамическая память

Во всех задачах этого блока используйте `malloc` и `free` из `<stdlib.h>`. Массивы фиксированной вместимости, массивы переменной длины, `new[]` и готовые контейнеры, включая `std::vector`, не допускаются. Размер выделяемой памяти вычисляйте по количеству элементов и размеру одного элемента, используя `sizeof`. Проверяйте результат каждого выделения перед обращением к памяти. После успешного выделения записывайте элементы до их чтения.

При ошибке выделения освободите все ранее выделенные в программе массивы, завершите программу с кодом 1 и ничего не выводите. Во всех остальных случаях завершите программу с кодом 0. После последнего использования каждого выделенного массива освободите его ровно один раз; после освобождения не обращайтесь к его элементам. Предполагается среда с 32-битным `int` и размером `size_t` не менее 32 бит; заданные ограничения позволяют представить размеры выделений.

Решения подходят для C и C++. В C результат `malloc` присваивается указателю без явного приведения типа; в C++ требуется явное преобразование из `void *` к нужному типу указателя.

### 3.1.1. Динамический массив

Дано количество чисел `N`, которое становится известно только после ввода. Выделите память для массива из `N` целых чисел, прочитайте в него данные и найдите минимальное значение, максимальное значение и среднее арифметическое.

Храните весь исходный массив в динамически выделенной памяти. Сначала прочитайте все его элементы, затем выполните расчёты. Сумму накапливайте в `long long`, а среднее вычисляйте в `double`. Выведите результат и освободите память.

**Формат входных данных**

В первой строке вводится целое число `N` — количество элементов. Во второй строке вводятся `N` целых чисел через пробел. Ограничения: $1 \le N \le 100000$, $-10^6 \le a_i \le 10^6$.

**Формат выходных данных**

Три значения в одной строке через пробел: минимальное число, максимальное число и среднее арифметическое. Среднее выведите с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 4<br>-2 4 0 6 | -2 6 2.000000 |
| 1<br>-7 | -7 -7 -7.000000 |
| 3<br>1000000 1000000 1000000 | 1000000 1000000 1000000.000000 |

### 3.1.2. Копия допустимых значений

Дан массив из N целых чисел и границы допустимого диапазона lower и upper. Создайте отдельный динамический массив, содержащий только значения из этого диапазона включительно. Сохраните порядок элементов и все повторяющиеся допустимые значения. Исходный массив не должен изменяться.

Реализуйте функцию `int *filter_values(const int source[], int count, int lower, int upper, int *result_count)`. Она сначала определяет количество подходящих элементов, затем выделяет память ровно для этого количества элементов, заполняет новый массив и возвращает указатель на него. Размер результата запишите через result_count, который указывает на существующую переменную типа int.

Если подходящих элементов нет, не вызывайте malloc: запишите в *result_count значение 0 и верните нулевой указатель. Если выделение памяти для непустого результата завершилось неудачно, запишите в *result_count значение $-1$ и верните нулевой указатель. Так вызывающий код сможет отличить пустой результат от ошибки выделения. Функция не должна выполнять ввод или вывод и не должна освобождать исходный массив.

В main динамически выделите исходный массив, прочитайте данные и вызовите filter_values. Выведите результат, затем освободите оба массива. Освобождение возвращённого массива является обязанностью вызывающего кода. При ошибке выделения соблюдайте общее правило блока.

**Формат входных данных**

В первой строке вводится целое число N. Во второй строке вводятся N целых чисел через пробел. В третьей строке вводятся два целых числа lower и upper. Ограничения: $1 \le N \le 100000$, $-10^6 \le a_i$, lower, upper $\le 10^6$, lower $\le$ upper.

**Формат выходных данных**

В первой строке выведите количество допустимых элементов. Во второй строке выведите эти элементы через пробел. Если допустимых элементов нет, первая строка должна содержать 0, а вторая должна быть пустой.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 6<br>-2 0 3 3 8 5<br>0 5 | 4<br>0 3 3 5 |
| 3<br>1 2 3<br>4 5 | 0 |
| 3<br>2 2 1<br>2 2 | 2<br>2 2 |

### 3.1.3. Объединение массивов

Даны два массива целых чисел. Создайте третий массив, содержащий сначала все элементы первого массива, затем все элементы второго. Порядок элементов внутри каждого исходного массива должен сохраниться; сортировать их не требуется.

Динамически выделите память для обоих исходных массивов. Реализуйте функцию `int *concat_arrays(const int first[], int first_count, const int second[], int second_count)`, которая выделяет отдельный массив из `first_count + second_count` элементов, копирует в него данные и возвращает указатель на результат. При ошибке выделения функция возвращает нулевой указатель. Функция не должна изменять или освобождать исходные массивы и не должна выполнять ввод или вывод.

В `main` прочитайте данные, вызовите функцию, выведите результат и освободите все три массива. Ответственность за освобождение возвращённого массива несёт вызывающий код. При ошибке выделения соблюдайте общее правило блока.

**Формат входных данных**

В первой строке вводится целое число `N` — размер первого массива. Во второй строке вводятся `N` целых чисел через пробел. В третьей строке вводится целое число `M` — размер второго массива. В четвёртой строке вводятся `M` целых чисел через пробел. Ограничения: $1 \le N, M \le 100000$, каждый элемент находится в диапазоне от $-10^6$ до $10^6$ включительно.

**Формат выходных данных**

Выведите `N + M` целых чисел в одной строке через пробел — элементы объединённого массива.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>1 4 2<br>2<br>9 -1 | 1 4 2 9 -1 |
| 1<br>0<br>1<br>0 | 0 0 |
| 2<br>-3 -3<br>3<br>5 0 5 | -3 -3 5 0 5 |

## 3.1.4. Динамический массив участников

В соревновании участвуют `N` человек. Для каждого заданы числовой идентификатор и результаты трёх этапов. Побеждает участник с наибольшей суммой результатов. При равенстве сумм побеждает участник с меньшим идентификатором.

Объявите структуру `struct Participant` с целочисленными полями `id`, `score1`, `score2` и `score3`. После чтения `N` динамически выделите память для массива из `N` таких структур и прочитайте в него всех участников. Размер выделения рассчитывайте через `sizeof` структуры или элемента массива, а не через сумму размеров её полей.

Реализуйте функцию `int best_participant(const struct Participant participants[], int count)`, которая возвращает индекс победителя с нуля, не изменяет массив и не выполняет ввод или вывод. В `main` вызовите функцию, выведите идентификатор победителя и сумму его результатов, затем освободите массив структур.

### Формат входных данных

В первой строке вводится целое число `N` — количество участников. Следующие `N` строк содержат по четыре целых числа: `id`, `score1`, `score2`, `score3`. Ограничения: $1 \le N \le 100000$, $1 \le id \le 10^9$, $0 \le score1, score2, score3 \le 100$. Идентификаторы участников различны.

### Формат выходных данных

Два целых числа в одной строке через пробел — идентификатор победителя и сумма его результатов на трёх этапах.

### Примеры

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>10 20 30 40<br>7 50 50 50<br>2 0 0 0 | 7 150 |
| 2<br>20 100 0 0<br>5 50 25 25 | 5 100 |
| 1<br>99 0 0 0 | 99 0 |

# Глава 8. ООП и структуры данных в C++

Все задачи этой дополнительной главы выполняются только на C++17. Отправляйте полную программу с функцией `main`. Ввод и вывод размещайте в `main`, если условие не требует другого. Закрытые поля изменяйте только через публичные операции класса. Соблюдайте указанные интерфейсы и требования к реализации: совпадение вывода само по себе не подтверждает их выполнение.

## Блок 8.1. Геттеры, сеттеры и инкапсуляция

### 8.1.1. Температура под контролем

Реализуйте класс `Temperature` с закрытым целочисленным полем, содержащим температуру в десятых долях градуса. Начальное значение — 0. Допустимы значения от 0 до 500 включительно.

Метод `bool set_raw(int value)` записывает допустимое значение и возвращает `true`. Если значение недопустимо, метод возвращает `false`, сохраняя прежнее состояние. Геттер `int get_raw() const` возвращает сохранённое значение, а `double get_celsius() const` — температуру в градусах Цельсия.

Создайте один объект и последовательно передайте ему все введённые значения через сеттер. После каждого вызова выведите успешность изменения и результаты обоих геттеров. Прямой доступ к полю из `main` запрещён.

**Формат входных данных**

В первой строке вводится число `N` — количество попыток изменения. Во второй строке вводятся `N` целых значений. Ограничения: $1 \le N \le 1000$, $-1000 \le value \le 1000$.

**Формат выходных данных**

Выведите `N` строк. Каждая содержит через пробел `OK` или `ERROR`, текущее значение в десятых долях градуса и значение в градусах Цельсия с шестью знаками после десятичной точки. Статус относится к текущему вызову сеттера, значения — к состоянию объекта после вызова.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>250 500 0 | OK 250 25.000000<br>OK 500 50.000000<br>OK 0 0.000000 |
| 3<br>-1 270 501 | ERROR 0 0.000000<br>OK 270 27.000000<br>ERROR 270 27.000000 |
| 2<br>0 0 | OK 0 0.000000<br>OK 0 0.000000 |

### 8.1.2. Границы диапазона

Реализуйте класс `Range` с закрытыми целочисленными полями нижней и верхней границ. Начальный диапазон — `[0, 100]`. Геттеры `int get_lower() const` и `int get_`

`upper() const` возвращают границы; метод `bool contains(int value) const` проверяет принадлежность включительному диапазону.

Метод `bool set_lower(int value)` изменяет нижнюю границу, только если она не больше текущей верхней. Метод `bool set_upper(int value)` изменяет верхнюю границу, только если она не меньше текущей нижней. Метод `bool set_bounds(int lower, int upper)` меняет обе границы одновременно при `lower` $\leq$ `upper`. При отказе любой сеттер возвращает `false` и сохраняет обе прежние границы. При успехе он возвращает `true`.

Обработайте команды: `LOWER value`, `UPPER value`, `BOUNDS lower upper`, `CONTAINS value`. Изменение диапазона выполняйте только через сеттеры.

**Формат входных данных**

В первой строке вводится `Q`. Следующие `Q` строк содержат команды указанного вида. Ограничения: `1` $\leq$ `Q` $\leq$ `1000`, все числовые аргументы находятся в диапазоне от $-$`10000` до `10000`. Для `BOUNDS` правильный порядок границ не гарантируется.

**Формат выходных данных**

Для каждой команды изменения выведите `OK` или `ERROR`, затем текущие нижнюю и верхнюю границы через пробел. Для `CONTAINS` выведите только `YES` или `NO`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 4<br>LOWER 10<br>UPPER 20<br>CONTAINS 15<br>CONTAINS 21 | OK 10 100<br>OK 10 20<br>YES<br>NO |
| 4<br>LOWER 101<br>BOUNDS 8 3<br>UPPER -1<br>CONTAINS 0 | ERROR 0 100<br>ERROR 0 100<br>ERROR 0 100<br>YES |
| 3<br>BOUNDS -5 -5<br>CONTAINS -5<br>LOWER -4 | OK -5 -5<br>YES<br>ERROR -5 -5 |

## Блок 8.2. Динамическая память, конструкторы и деструкторы

В задачах блока реализуйте владение массивом вручную через `new[]` и `delete[]`. Готовые контейнеры и умные указатели не допускаются. Деструктор должен освобождать принадлежащую объекту память. Для простого варианта запретите копирующий конструктор и копирующее присваивание через `= delete`; перемещение реализовывать не требуется. Объекты создавайте как локальные переменные, чтобы деструкторы вызывались при выходе из области видимости. Конструкторы и деструкторы не должны печатать служебные сообщения.

При неудаче выделения перехватите `std::bad_alloc` в `main`, выведите `MEMORY ERROR` на отдельной строке и завершите программу с кодом 1. Уже созданные объекты должны корректно освободить свои ресурсы. В обычном случае код завершения — 0.

### 8.2.1. Буфер чисел

Реализуйте класс `IntBuffer` с закрытыми полями размера и указателя на динамический массив. Конструктор `explicit IntBuffer(int size)` выделяет массив указанного размера и инициализирует все элементы нулями; размер положителен. Деструктор освобождает массив.

Предусмотрите геттер `int size() const`, метод `bool set(int index, int value)` и метод `bool get(int index, int &value) const`. При допустимом индексе методы возвращают `true`; `set` изменяет элемент, `get` записывает элемент в выходной параметр. При недопустимом индексе методы возвращают `false`, не изменяя массив и, для `get`, выходной параметр.

Метод `long long sum() const` возвращает сумму элементов. Обработайте команды `SET index value`, `GET index`, `SUM`. Все обращения к массиву из `main` должны выполняться через методы.

**Формат входных данных**

В первой строке вводятся размер `N` и количество команд `Q`. Следующие `Q` строк содержат команды. Ограничения: $1 \le N \le 100000$, $1 \le Q \le 1000$, $-1 \le index \le N$, $-10^6 \le value \le 10^6$. Индексация начинается с нуля.

**Формат выходных данных**

Для `SET` выведите `OK` или `ERROR`. Для `GET` — значение элемента либо `ERROR`. Для `SUM` — сумму элементов. Каждый ответ выведите с новой строки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3 5<br>SUM<br>SET 1 7<br>GET 1<br>GET 0<br>SUM | 0<br>OK<br>7<br>0<br>7 |
| 2 4<br>SET -1 8<br>GET 2<br>SET 0 -5<br>SUM | ERROR<br>ERROR<br>OK<br>-5 |
| 1 4<br>SET 0 10<br>SET 0 20<br>GET 0<br>SUM | OK<br>OK<br>20<br>20 |

### 8.2.2. Матрица с собственным хранилищем

Реализуйте класс `Matrix`, который хранит число строк, число столбцов и указатель на один динамический массив из `rows` × `columns` целых элементов. Конструктор `Matrix(int rows, int columns)` выделяет массив и заполняет его нулями. Деструктор освобождает память. Оба размера положительны.

Предусмотрите геттеры `int rows() const`, `int columns() const`, методы `bool set(int row, int column, int value)`, `bool get(int row, int column, int &value) const` и `long long row_sum(int row) const`. Храните строки последовательно в одном массиве. При недопустимых индексах `set` и `get` возвращают `false`, сохраняя состояние и выходной параметр. Метод `row_sum` вызывается только для допустимой строки.

В `main` создайте матрицу, заполните её через `set`, выведите суммы строк, затем выполните запросы чтения через `get`. Запретите копирование объекта, как указано в общих требованиях блока.

**Формат входных данных**

В первой строке вводятся `R` и `C`. Следующие `R` строк содержат по `C` целых чисел. Затем вводится `Q`, после него — `Q` строк с парами `row column`. Ограничения: $1 \le$ `R, C` $\le 200$, $-10^6 \le a_{ij} \le 10^6$, $1 \le$ `Q` $\le 1000$, $-1 \le$ `row` $\le$ `R`, $-1 \le$ `column` $\le$ `C`. Индексация начинается с нуля.

**Формат выходных данных**

В первой строке выведите суммы всех строк через пробел. Затем для каждого запроса выведите значение элемента либо `ERROR`, каждый ответ с новой строки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 2 2<br>1 2<br>3 4<br>2<br>0 1<br>1 0 | 3 7<br>2<br>3 |
| 1 1<br>-7<br>3<br>0 0<br>-1 0<br>0 1 | -7<br>-7<br>ERROR<br>ERROR |
| 2 3<br>-1 0 1<br>1000000 1000000 1000000<br>1<br>1 2 | 0 3000000<br>1000000 |

## Блок 8.3. Наследование и полиморфизм

### 8.3.1. Правила тревоги

Объявите абстрактный класс `AlarmRule` с публичным виртуальным деструктором и чисто виртуальным методом `bool is_alarm(int value) const`. Создайте два производных класса: `AboveThreshold`, для которого тревога возникает при `value` $\ge$ `threshold`, и `OutsideRange`, для которого тревога возникает при `value < lower` или `value > upper`. Границы и порог задаются конструкторами и хранятся в закрытых полях. Переопределённые методы пометьте `override`.

Реализуйте функцию `int count_alarms(const int values[], int count, const AlarmRule &rule)`. Она подсчитывает тревожные значения, вызывая виртуальный метод правила. Внутри функции не проверяйте конкретный тип правила.

В `main` прочитайте значения и настройки, создайте оба правила и вызовите одну и ту же функцию сначала для порогового правила, затем для диапазонного.

**Формат входных данных**

В первой строке вводится `N`, во второй — `N` целых значений. В третьей строке вводятся `threshold lower upper`. Ограничения: $1 \le N \le 1000$, все значения и настройки находятся в диапазоне от $-10000$ до 10000, $\texttt{lower} \le \texttt{upper}$.

**Формат выходных данных**

Два целых числа через пробел — количество тревог для `AboveThreshold` и для `OutsideRange` соответственно.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 5<br>10 20 30 40 50<br>30 20 40 | 3 2 |
| 3<br>5 5 5<br>5 5 5 | 3 0 |
| 4<br>-10 0 10 20<br>100 -10 20 | 0 0 |

## 8.3.2. Разные фигуры — общий интерфейс

Объявите абстрактный класс `Shape` с публичным виртуальным деструктором и чисто виртуальным методом `double area() const`. Создайте классы `Rectangle` и `Circle`, хранящие размеры в закрытых полях, заданных конструкторами. Площадь прямоугольника равна произведению сторон; площадь круга вычисляется с константой `PI = 3.141592653589793`.

Прочитайте описания фигур и сохраните объекты в `std::vector<std::unique_ptr<Shape>>`. Для создания используйте `std::make_unique`; подключите необходимые заголовки. Последовательно вызовите `area()` через общий интерфейс и найдите сумму площадей. Выбор формулы по текстовому типу допускается только при создании объекта; при вычислении площадей используйте виртуальные вызовы. Ресурсы должны освобождаться автоматически.

**Формат входных данных**

В первой строке вводится `N`. Следующие `N` строк имеют вид `RECT width height` или `CIRCLE radius`. Ограничения: $1 \le N \le 1000$, $0 < \texttt{width, height, radius} \le 1000$. Размеры вещественные, содержат не более двух знаков после десятичной точки.

**Формат выходных данных**

Сначала выведите N площадей в порядке ввода, каждую с новой строки. В следующей строке выведите их сумму. Все значения выводите с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 2<br>RECT 2 3<br>CIRCLE 1 | 6.000000<br>3.141593<br>9.141593 |
| 1<br>RECT 0.50 4 | 2.000000<br>2.000000 |
| 2<br>CIRCLE 2<br>CIRCLE 1 | 12.566371<br>3.141593<br>15.707963 |

## Блок 8.4. Перегрузки, шаблоны и пространства имён

### 8.4.1. Ограничитель для двух типов

В пространстве имён control реализуйте две перегрузки функции clamp_value: одну с тремя аргументами и результатом типа int, другую — типа double. Функция возвращает нижнюю границу, если значение меньше неё, верхнюю — если оно больше неё, иначе само значение. Границы включительны и заданы в правильном порядке.

Обработайте запросы INT value lower upper и DOUBLE value lower upper, выбирая нужную перегрузку по типу прочитанных переменных. В main вызывайте функции через полное имя control::clamp_value. Готовые функции ограничения значений не допускаются.

**Формат входных данных**

В первой строке вводится Q. Следующие Q строк содержат запросы. Ограничения: $1 \le Q \le 1000$, все числовые аргументы находятся в диапазоне от $-10000$ до 10000, $lower \le upper$. Для INT аргументы целые; для DOUBLE — вещественные с не более чем двумя знаками после десятичной точки.

**Формат выходных данных**

Для каждого запроса выведите ограниченное значение с новой строки. Для INT — целое число, для DOUBLE — число с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>INT -1 0 10<br>DOUBLE 12.5 0 10<br>INT 5 0 10 | 0<br>10.000000<br>5 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 2<br>DOUBLE -2 -2 -2<br>INT 100 7 7 | -2.000000<br>7 |
| 2<br>INT -10 -20 -5<br>DOUBLE 1.25 0.5 2.5 | -10<br>1.250000 |

### 8.4.2. Шаблон диапазона

В пространстве имён `control` реализуйте шаблон класса `Range<T>`. Закрытые поля нижней и верхней границ имеют тип `T` и задаются конструктором `Range(T lower, T upper)`. Предусмотрите методы `T get_lower() const`, `T get_upper() const`, `bool contains(T value) const` и `T clamp(T value) const`. Последний метод ограничивает значение включительным диапазоном.

Прочитайте границы и запросы, создайте объект `control::Range<int>` или `control::Range<double>` в зависимости от указанного типа. Для каждого значения определите принадлежность диапазону и результат ограничения. Реализуйте один шаблон; отдельные копии класса для `int` и `double` не допускаются.

**Формат входных данных**

В первой строке вводится слово `INT` или `DOUBLE`. Во второй строке — `lower upper`. В третьей строке — `Q`, в четвёртой — `Q` значений. Ограничения: $1 \le Q \le 1000$, все числа находятся в диапазоне от $-10000$ до $10000$, $lower \le upper$. Для `INT` числа целые; для `DOUBLE` — вещественные с не более чем двумя знаками после десятичной точки.

**Формат выходных данных**

Выведите `Q` строк. Каждая содержит `YES` или `NO` — принадлежность исходного значения диапазону — и ограниченное значение через пробел. Для `INT` выводите целое число; для `DOUBLE` — число с шестью знаками после десятичной точки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| INT<br>0 10<br>3<br>-1 5 11 | NO 0<br>YES 5<br>NO 10 |
| DOUBLE<br>0.5 1.5<br>3<br>0.5 1.25 2 | YES 0.500000<br>YES 1.250000<br>NO 1.500000 |
| INT<br>7 7<br>2<br>7 6 | YES 7<br>NO 7 |

# Блок 8.5. Стек

## 8.5.1. Стек команд

Реализуйте класс `IntStack`, работающий по принципу LIFO: последним добавленный элемент извлекается первым. Храните элементы в закрытом массиве фиксированной вместимости 1000 и задайте рабочую вместимость `capacity` конструктором. Использование готовых контейнеров, включая `std::stack` и `std::vector`, запрещено.

Предусмотрите методы `bool push(int value)`, `bool pop(int &value)`, `bool top(int &value) const`, `int size() const`, `bool empty() const`. `pop` удаляет верхний элемент и записывает его в выходной параметр; `top` только читает верхний элемент. При переполнении или обращении к пустому стеку операция возвращает `false`, не изменяя стек и выходной параметр. При успехе — `true`.

Обработайте команды `PUSH value`, `POP`, `TOP`, `SIZE`, `EMPTY`.

**Формат входных данных**

В первой строке вводятся `capacity` и `Q`. Следующие `Q` строк содержат команды. Ограничения: $1 \le$ `capacity` $\le 1000$, $1 \le$ `Q` $\le 10000$, $-10^6 \le$ `value` $\le 10^6$.

**Формат выходных данных**

Для `PUSH` выведите `OK` или `FULL`; для `POP` и `TOP` — значение либо `EMPTY`; для `SIZE` — размер стека; для `EMPTY` — `YES` или `NO`. Каждый ответ выведите с новой строки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 2 7<br>PUSH 10<br>PUSH 20<br>TOP<br>POP<br>TOP<br>SIZE<br>EMPTY | OK<br>OK<br>20<br>20<br>10<br>1<br>NO |
| 1 6<br>POP<br>PUSH 7<br>PUSH 8<br>POP<br>TOP<br>EMPTY | EMPTY<br>OK<br>FULL<br>7<br>EMPTY<br>YES |
| 2 5<br>PUSH -5<br>POP<br>PUSH 0<br>TOP<br>SIZE | OK<br>-5<br>OK<br>0<br>1 |

### 8.5.2. Правильные скобки

Дана строка из круглых, квадратных и фигурных скобок. Проверьте, является ли последовательность правильной: каждой закрывающей скобке должна соответствовать последняя ещё не закрытая открывающая скобка того же вида; после обработки строки незакрытых скобок не должно остаться.

Реализуйте собственный класс `BracketStack` поверх массива `char` вместимости 10000 с операциями добавления, извлечения и проверки пустоты. Используйте его для проверки последовательности. Готовые контейнеры для хранения стека запрещены. Саму входную строку разрешено хранить в `std::string`.

**Формат входных данных**

Одна непустая строка длиной от 1 до 10000 символов. Каждый символ — один из `(`, `)`, `[`, `]`, `{`, `}`. Пробельных символов нет.

**Формат выходных данных**

Выведите `YES`, если скобочная последовательность правильная, иначе `NO`.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| `([]){}` | `YES` |
| `([)]` | `NO` |
| `(()` | `NO` |

## Блок 8.6. Очередь и кольцевой буфер

### 8.6.1. Кольцевая очередь команд

Реализуйте класс `IntQueue`, работающий по принципу FIFO: первым добавленный элемент извлекается первым. Храните элементы в закрытом массиве вместимости 1000; рабочую вместимость задайте конструктором. Используйте кольцевую индексацию. Добавление и извлечение должны выполняться за `O(1)`, без сдвига элементов массива. Готовые контейнеры запрещены.

Предусмотрите методы `bool push(int value)`, `bool pop(int &value)`, `bool front(int &value) const`, `int size() const`, `bool empty() const`. При добавлении в заполненную очередь и чтении или извлечении из пустой операция возвращает `false`, сохраняя состояние и выходной параметр. В остальных случаях — `true`.

Обработайте команды `PUSH value`, `POP`, `FRONT`, `SIZE`, `EMPTY`.

**Формат входных данных**

В первой строке вводятся `capacity` и `Q`. Следующие `Q` строк содержат команды. Ограничения: $1 \le$ `capacity` $\le 1000$, $1 \le$ `Q` $\le 10000$, $-10^6 \le$ `value` $\le 10^6$.

**Формат выходных данных**

Для `PUSH` выведите `OK` или `FULL`; для `POP` и `FRONT` — значение либо `EMPTY`; для `SIZE` — количество элементов; для `EMPTY` — `YES` или `NO`. Каждый ответ выведите с новой строки.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3 7<br>PUSH 10<br>PUSH 20<br>FRONT<br>POP<br>FRONT<br>SIZE<br>EMPTY | OK<br>OK<br>10<br>10<br>20<br>1<br>NO |
| 1 6<br>POP<br>PUSH 7<br>PUSH 8<br>POP<br>FRONT<br>EMPTY | EMPTY<br>OK<br>FULL<br>7<br>EMPTY<br>YES |
| 2 8<br>PUSH 1<br>PUSH 2<br>POP<br>PUSH 3<br>POP<br>POP<br>SIZE<br>EMPTY | OK<br>OK<br>1<br>OK<br>2<br>3<br>0<br>YES |

## 8.6.2. Последние измерения

Реализуйте класс `RecentMeasurements`, который хранит не более `capacity` последних целочисленных измерений в кольцевом буфере. Конструктор задаёт вместимость; исходно буфер пуст. Метод `void add(int value)` добавляет измерение, а при заполненном буфере заменяет самое старое. Новое значение становится самым свежим.

Предусмотрите методы `int size() const`, `int get(int index) const` и `double average() const`. Метод `get` использует логические индексы с нуля: 0 соответствует самому старому сохранённому измерению. Он вызывается только с допустимым индексом. Среднее вычисляйте по всем сохранённым измерениям; на пустом буфере `average()` возвращает 0.0.

Используйте закрытый массив вместимости 1000 и кольцевую индексацию; готовые контейнеры и сдвиг всех элементов при добавлении запрещены. После каждого нового измерения выведите количество сохранённых значений и среднее. В конце выведите содержимое буфера от старого к новому.

**Формат входных данных**

В первой строке вводятся `capacity` и `N`. Во второй — `N` целых измерений. Ограничения: $1 \le$ `capacity` $\le 1000$, $1 \le$ `N` $\le 10000$, $-10^6 \le$ `value` $\le 10^6$.

**Формат выходных данных**

После каждого добавления выведите отдельную строку: текущий размер и среднее с шестью знаками после десятичной точки через пробел. В последней строке выведите все сохранённые измерения через пробел в порядке от старого к новому.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3 5<br>10 20 30 40 50 | 1 10.000000<br>2 15.000000<br>3 20.000000<br>3 30.000000<br>3 40.000000<br>30 40 50 |
| 1 3<br>-5 0 7 | 1 -5.000000<br>1 0.000000<br>1 7.000000<br>7 |
| 4 2<br>1 2 | 1 1.000000<br>2 1.500000<br>1 2 |

## Блок 8.7. Стандартные контейнеры

### 8.7.1. Изменяемый список

Храните список целых чисел в `std::vector<int>`. Исходно список пуст. Обработайте команды `PUSH value`, `INSERT index value`, `ERASE index`, `GET index`, `SIZE`. `INSERT` вставляет перед элементом с указанным индексом; индекс, равный текущему размеру, означает добавление в конец. `ERASE` удаляет элемент; `GET` читает его. Индексация начинается с нуля.

Для `INSERT` допустимы индексы от 0 до размера включительно, для `ERASE` и `GET` — от 0 до размера минус один. При недопустимом индексе выведите `ERROR` и сохраните список. Все изменения выполняйте средствами `std::vector`.

**Формат входных данных**

В первой строке вводится `Q`. Следующие `Q` строк содержат команды. Ограничения: $1 \le$ `Q` $\le 10000$, $-1 \le$ `index` $\le 10000$, $-10^6 \le$ `value` $\le 10^6$.

**Формат выходных данных**

Для успешных `PUSH`, `INSERT`, `ERASE` выведите `OK`. Для `GET` — значение, для `SIZE` — текущий размер. При ошибке индекса выведите `ERROR`. Ответ на каждую команду — отдельная строка. После всех ответов выведите итоговый список в одной строке через пробел; для пустого списка выведите пустую строку.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 6<br>PUSH 10<br>PUSH 30<br>INSERT 1 20<br>GET 1<br>ERASE 0<br>SIZE | OK<br>OK<br>OK<br>20<br>OK<br>2<br>20 30 |
| 4<br>GET 0<br>INSERT -1 5<br>ERASE 0<br>SIZE | ERROR<br>ERROR<br>ERROR<br>0<br><br>Последняя строка вывода пустая: контейнер не содержит элементов. |
| 5<br>INSERT 0 7<br>INSERT 1 9<br>INSERT 3 11<br>ERASE 1<br>GET 0 | OK<br>OK<br>ERROR<br>OK<br>7<br>7 |

## 8.7.2. Частоты значений

Дана последовательность целых чисел. С помощью `std::map<int, int>` подсчитайте количество появлений каждого значения. Выведите различные значения в порядке возрастания вместе с их частотами. Массив счётчиков, сортировка исходных данных и другие ассоциативные контейнеры вместо `std::map` не допускаются.

**Формат входных данных**

В первой строке вводится `N`. Во второй — `N` целых чисел. Ограничения: $1 \le$ `N` $\le 100000$, $-10^9 \le$ `value` $\le 10^9$.

**Формат выходных данных**

В первой строке выведите количество различных значений. Затем для каждого значения в порядке возрастания выведите отдельную строку: значение и количество его появлений через пробел.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 7<br>3 -1 3 2 -1 3 0 | 4<br>-1 2<br>0 1<br>2 1<br>3 3 |
| 4<br>5 5 5 5 | 1<br>5 4 |

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 3<br>1000000000 -1000000000 0 | 3<br>-1000000000 1<br>0 1<br>1000000000 1 |

### 8.7.3. Множество идентификаторов

Храните уникальные целочисленные идентификаторы в `std::set<int>`. Исходно множество пусто. Обработайте команды `ADD value`, `REMOVE value`, `HAS value`, `SIZE`. Повторное добавление не создаёт новый элемент; удаление отсутствующего значения не меняет множество.

Для `ADD` определите, было ли значение добавлено впервые; для `REMOVE` — был ли элемент удалён. Используйте операции `std::set`; хранение всех идентификаторов в массиве или `std::vector` вместо множества запрещено.

**Формат входных данных**

В первой строке вводится `Q`. Следующие `Q` строк содержат команды. Ограничения: $1 \le Q \le 100000$, $0 \le value \le 10^9$.

**Формат выходных данных**

Для `ADD` выведите `ADDED` или `EXISTS`; для `REMOVE` — `REMOVED` или `NOT FOUND`; для `HAS` — `YES` или `NO`; для `SIZE` — текущий размер множества. Каждый ответ — отдельная строка. После всех ответов выведите оставшиеся идентификаторы в порядке возрастания через пробел; если множество пусто, выведите пустую строку.

**Примеры**

| ВХОДНЫЕ ДАННЫЕ | ВЫХОДНЫЕ ДАННЫЕ |
|---|---|
| 6<br>ADD 7<br>ADD 3<br>ADD 7<br>HAS 3<br>REMOVE 7<br>SIZE | ADDED<br>ADDED<br>EXISTS<br>YES<br>REMOVED<br>1<br>3 |
| 4<br>REMOVE 5<br>HAS 5<br>ADD 5<br>REMOVE 5 | NOT FOUND<br>NO<br>ADDED<br>REMOVED<br><br>Последняя строка вывода пустая: контейнер не содержит элементов. |
| 5<br>ADD 1000000000<br>ADD 0<br>SIZE<br>HAS 1<br>REMOVE 1 | ADDED<br>ADDED<br>2<br>NO<br>NOT FOUND<br>0 1000000000 |

# Источники


1. Arm. ELF for the Arm Architecture (AAELF32) — общая структура ELF и особенности, зависящие от архитектуры Arm.
2. STMicroelectronics. STM32CubeIDE user guide, UM2609 — стартовые файлы, сценарии компоновки и размещение программы в памяти.
3. ISO/IEC JTC1/SC22/WG14. N1570, Committee Draft, 12 April 2011, открытый проект стандарта C11.
4. ISO/IEC JTC1/SC22/WG21. N4659, открытый проект C++17, раздел [dcl.ref], References: ссылки — правила ссылок, используемые в разделе 2.2.
5. ISO/IEC JTC1/SC22/WG21. N4659, открытый проект C++17, разделы [dcl.constexpr], constexpr, и [expr.const], константные выражения — правила, используемые в разделе 3.3.
6. GNU Compiler Collection. Volatiles — особенности обращений к volatile в GCC.
7. GNU Compiler Collection. Overall Options — этапы сборки и остановка после выбранного этапа.
8. GNU Binutils. Документация инструментов — nm, readelf, objdump, size и objcopy.
9. GNU ld. Руководство — сценарии компоновки, MEMORY, SECTIONS, LMA, ENTRY, map-файл и удаление секций.
10. STMicroelectronics. DS10314, STM32F411xC/xE datasheet — объём и организация памяти выбранного МК.
11. STMicroelectronics. RM0383, STM32F411xC/xE reference manual — карта памяти и выбор режима загрузки.
12. STMicroelectronics. PM0214, STM32 Cortex-M4 programming manual — сброс, MSP, таблица векторов и Thumb-состояние.
13. STMicroelectronics. CMSIS Device F4, startup_stm32f411xe.s, commit 3b64b7b0b546cefe84f13c65508126eea58b97d8 — стартовый файл учебного проекта.
14. Arm GNU Toolchain — кросс-компилятор и инструменты; проект проверен с 14.2.Rel1.
15. OpenOCD. GDB and OpenOCD — подключение GDB, загрузка и исследование программы.
16. ISO/IEC JTC1/SC22/WG21. N4659, открытый проект C++17 — классы, доступ к членам, конструкторы и деструкторы, наследование и виртуальные функции, пространства имён, перегрузки, шаблоны, выделение и освобождение памяти, обработка исключений, `unique_ptr`, `make_unique`, контейнеры `vector`, `map`, `set`, потоки ввода и вывода и манипуляторы форматирования. Ссылки на конкретные разделы приведены в теоретической части.
17. GNU Compiler Collection. Invoking G++ — компиляция C++ и подключение его стандартной библиотеки.
18. Microsoft. Using GCC with MinGW — настройка Visual Studio Code для сборки, запуска и отладки программ на Windows; материал используется в главе 1.

19. GNU C Library. Formatted Input Basics и Formatted Output Basics — основы форматированного ввода и вывода, используемые в разделе 1.6.

20. Яндекс Образование. Основные виды сортировок и примеры их реализации — материал к заданиям 2.1.10 «Пузырьковая сортировка» и 2.1.12 «Быстрая сортировка».

21. Open-JS. Бинарный поиск. Хабр — материал к заданию 2.1.11; примеры на JavaScript адаптируются к C или C++ согласно условию задания.

22. Algorithmica. Алгоритм Евклида — вычисление НОД для сокращения дробей в задании 2.3.5 «Сложение дробей».

23. Open-JS. Сложность алгоритмов. Разбор Big O. Хабр — материал об асимптотической оценке сложности, включённый в задание 2.1.10 «Пузырьковая сортировка».