Язык C++

Андрей Подшивалов

2025 г.

Содержание

1 Начало

1.1 О структуре

Некоторые факты изложены в выделенных секциях. Ниже описано их предназначение.

Математическая справка. Иногда в информатике требуются знания из математики, которые в курсах алгебры, геометрии и спецмата ещё не пройдены к моменту изучения в информатике. Такие факты приводятся в этой секции.

Стилистическая заметка. Соблюдение этих правил настоятельно рекомендуется для поддержания читабельности кода и удобства его понимания. Но, вообще говоря, соблюдение данных правил не обязательно.

Важно! У C++ есть свои странности, о которых важно помнить, чтобы Ваша программа работала так, как Вы хотите.

Замечание. В данной секции приводятся интересные факты или обращается внимание на некоторые факты.

Задача. Так обозначаются задачи, формулируемые для последующего разбора.

Кроме того, *курсивом* выделены новые термины, а важные, по мнению автора, факты подчёркнуты.

В коде после `//` указывается вывод или возвращаемое значение инструкции в данной строке.

В случае ошибок, опечаток, предложений по оформлению и т.д. пишите @cpp_is_ok в Телеграмм.

В случае вопросов по содержанию или по задачам пишите Боту помощи @inf54bot.

1.2 Шаблон

Ниже приведён стандартный код на C++:

```
#include <iostream>

using namespace std;

int main() {

}
```

Весь код, который мы будем писать в первом полугодии 8 класса, будет находиться внутри фигурных скобок. Пока будем использовать данный шаблон, не задумываясь, что значат все эти инструкции. Потом мы узнаем об их назначении. Часто в примерах далее мы будем опускать шаблон. В таком случае предполагается, что приведённый код находится внутри фигурных скобок.

Стилистическая заметка. Код внутри фигурных скобок принято писать с отступом в 4 пробела в начале каждой строки. Скорее всего, редактор кода поможет Вам с поддержанием этого отступа.

1.3 Переменные

Переменная — это «коробка» для хранения чего-либо. У переменной есть:

1. **Тип**. Каждая коробка подходит только для одного типа содержимого. Пока мы будем работать с целыми числами — это один из возможных типов содержимого.
2. **Значение**. Это и есть то, что лежит в коробке. Например, там может лежать число 54.
3. **Имя**. Некоторая последовательность символов, уникальная для каждой переменной. По ней в переменную можно записывать значения и получать значения.

Каждая инструкция в C++ заканчивается точкой с запятой (`;`).

Чтобы объявить переменную (создать коробку), нужно написать сначала тип переменной, затем её имя:

`Type name;`,

где `Type` — тип переменной, с которыми мы будем знакомиться постепенно, а `name` — имя переменной.

Например, инструкция

```
int a;
```

создаёт переменную с именем `a` типа целое число (`int` от англ. integer — целое). В названии можно использовать любые английские буквы (как заглавные, так и строчные), нижнее подчеркивание (_) и цифры, но название не может начинаться с цифры. Название переменной может содержать любое количество символов. Название переменной не должно совпадать с ключевыми словами (с ними мы будем постепенно знакомиться в следующих главах) и типами данных. В таблице ниже приведены примеры корректных и некорректных имён переменных.

Корректные	Некорректные
a2	2a
number_of_letters	number-of-letters
numberOfLetters	number of letters
int_	int

Стилистическая заметка. Хорошо давать переменным имена, из которых следует содержимое переменных. Исключением является размер входных данных: если в условии сказано, что программе будет передано некоторое количество целых чисел, то это количество обычно обозначается n (или m).

Чтобы *присвоить значение* переменной (положить что-либо в коробку, навсегда утратив предыдущее её содержимое), нужно написать через равно имя переменной и новое её значение. Например, чтобы присвоить уже объявленной переменной `a` число 54, нужно написать:

```
a = 54;
```

Таким образом, чтобы объявить переменную `letter_num` и присвоить ей значение 5, нужно написать:

```
int letter_num;
letter_num = 5;
```

Вместо этого есть более короткая запись:

```
int letter_num = 5;
```

Можно объявлять несколько переменных одного типа в одной строке:

```
int a, b = 5, c = 4;
```

Эта строка объявляет переменные `a`, `b` и `c`. Переменным `b` и `c` заданы значения (5 и 4 соответственно). Переменная `a` создана, но *не инициализирована*. Это значит, что в ней может храниться любое значение, пока оно не будет задано явно.

Используя `равно`, можно сохранять значение одной переменной в другую. Например, если в переменной `b` хранится значение 5, то после инструкции

```
int a = b;
```

будет создана переменная `a`, в которой тоже будет храниться число 5.

Когда некоторое число используется в коде несколько раз, то хорошим стилем написания кода является определение *константы*. Это переменная, значение которой может быть присвоено только при создании. При попытке далее в программе изменить значение константы программа не может быть собрана, что гарантирует сохранность одного значения на протяжении всего выполнения кода.

Чтобы объявить константу, перед названием типа надо написать `const`:

```
const int SYSTEM_BASE = 10;
```

Стилистическая заметка. Для именования констант используют только заглавные буквы и нижние подчёркивания, как в примере выше.

Стилистическая заметка. Любую константу (кроме 0 и 1) следует явно объявлять как константы.

1.4 Арифметические операции

В данном параграфе мы научимся совершать арифметические операции — сложение, вычитание, умножение, деление, остаток от деления. В таблице ниже приведены обозначения этих операций в C++:

сложение	+
вычитание	-
умножение	*
деление	/
остаток от деления	%

Эти операции можно совершать как с числами, так и с переменными. В одном выражении можно использовать и числа, и переменные. Порядок действий по умолчанию — как в математике. Приоритет остатка от деления такой же, как у деления. Чтобы повлиять на порядок действий, можно использовать круглые скобки.

`+`, `-`, `*`, `/` — это *бинарные операторы*. Это значит, что они совершают некоторое действие с двумя *операндами* — теми объектами, которые они разделяют. На самом деле, `=` — тоже бинарный оператор, называемый *оператором присваивания*.

Следующий код сохраняет числа 5 и 4 в переменные `a` и `b`, а затем сохраняет их сумму в переменную `c`.

```cpp
#include<iostream>

using namespace std;

int main() {
    int a = 5, b = 4;
    int c = a + b;
}
```

Стилистическая заметка. Хорошо ставить пробелы с обеих сторон от оператора, как в примере выше. Также пробел ставится после запятой, но не до неё.

Математическая справка. *Разделить* целое число a на натуральное число b *с остатком* значит представить a в виде $qb + r$, где q и r целые, и $0 \le r < b$. Тогда число q называется *неполным частным*, а число r *остатком* при делении числа a на число b.

Операция деления (`/`) над целыми числами возвращает именно **неполное частное**. Например, `5/2` равно 2.

Важно! При делении отрицательного числа на положительное C++ не соблюдает правила математики. Он делает операции так, будто это положительное число, а затем добавляет неполному частному и остатку знак минус. Например, $(-5)\%3 = -2$, $(-5)/3 = -1$, так как $-5 = (-3) \cdot (-1) + (-2)$.

Если Вам нужен остаток от деления, как в математике, для деления a на b можно использовать следующую конструкцию:

```cpp
((a % b) + b) % b
```

Кроме того, есть *унарный оператор* $-$. Это значит, что он совершает операцию, имеющую только один операнд (который идет после оператора). При таком использовании он меняет знак у операнда. Например, после выполнения кода

```cpp
int a = 5;
int b = -a;
```

в переменной `b` будет храниться число -5.

1.5 Обработка цифр числа

Научимся обрабатывать цифры числа. Для этого узнаем, как совершать два типа операций:

1. Удаление последней цифры
2. Получение последней цифры

Математическая справка. Если над некоторым набором переменных стоит палочка, это значит, что рассматривается число, цифры которого — значения переменных. Например, если $a = 5, b = 4$, то $\overline{ab} = 54$, а $ab = a \cdot b = 20$. Палочка используется для того, чтобы не путать запись цифр числа и перемножение.

Рассмотрим некоторое натуральное число $a = \overline{a_n a_{n-1} \dots a_0}$. Разложим его на разрядные слагаемые и проведём некоторые преобразования: $a = a_n \cdot 10^n + a_{n-1} \cdot 10^{n-1} + \dots + a_0 = 10(a_n \cdot 10^{n-1} + a_{n-1} \cdot 10^{n-2} + \dots + a_1) + a_0 = 10\overline{a_n a_{n-1} \dots a_1} + a_0$. $a_0 < 10$, и $\overline{a_n a_{n-1} \dots a_1}$ и a_0 целые, поэтому по определению деления с остатком a_0 — остаток, $\overline{a_n a_{n-1} \dots a_1}$ — неполное частное при делении на 10.

Тогда если взять остаток от деления некоторого числа на 10, то мы получим его последнюю цифру, а если поделить его нацело на 10, то удалим её.

1.6 Про тип данных `long long`

В п. 1.3 мы познакомились с типом данных `int`. На самом деле, он умеет вмещать в себя только числа в диапазоне $[-2^{31}; 2^{31} - 1]$ (примерно от $-2 \cdot 10^9$ до $2 \cdot 10^9$). Но часто требуется работать с б'ольшими числами. Тип данных `long long` позволяет работать с числами в диапазоне $[-2^{63}; 2^{63} - 1]$ (примерно от $-9 \cdot 10^{18}$ до $9 \cdot 10^{18}$). Операции над типом `long long` совершаются так же, как и над типом `int`. Но помните, что при перемножении двух чисел типа `int` результат будет считаться тоже в типе int. Поэтому следующий код **содержит ошибку**:

```cpp
int a = 1000000000;
long long b = a * a;
```

Результат перемножения считается в типе `int`, в который не помещается. Есть несколько возможных способов исправления этой ошибки:

```cpp
long long a = 1000000000;
long long b = a * a;
```

или

```
int a = 1000000000;
long long b = 1ll * a * a;
```

Обратите внимание, что если после числа написано `ll`, то оно имеет тип `long long`. Если умножить `int` на `long long`, то получится `long long`. Поэтому первое умножение в коде выше приведет `a` к типу `long long`.

1.7 Ввод-вывод

Обычно от программы требуется какое-либо взаимодействие с пользователем. Поэтому в C++, как и во всех других языках программирования, реализован ввод с клавиатуры и вывод на экран. Доступ к вводу осуществляется через `cin` (от англ. console input — консольный ввод) — особый объект. Чтобы считать число, например, в переменную `a`, нужно написать следующую инструкцию:

```
cin >> a;
```

(Помните, что перед работой с переменной её надо обязательно объявить!)

Кроме того, можно считывать несколько чисел одной инструкцией:

```
cin >> a >> b >> c;
```

При этом при вводе числа могут разделяться как переводами строк (`Enter`), так и пробелами.

Для вывода используется другой объект — `cout` (от англ. console output — консольный вывод). Чтобы вывести содержимое переменной `a`, нужно написать:

```
cout << a;
```

Выводить можно не только переменные, но и числа. Также, как и со вводом, можно выводить несколько объектов за раз. Но учитывайте, что если вывести сначала число 1, а затем число 5, то на экране будет выведено 15. Пробелы или переводы строк нужно выводить самостоятельно. Чтобы вывести пробел, выведите `' '`. Чтобы вывести перевод строки, выведите `endl` или `'\n'`. Следует помнить, что `'\n'` работает быстрее, чем `endl`. Пример:

```
cout << 5 << ' ' << 6 << 7 << '\n' << 8 + 3 << endl << 9;
```

Эта строка выведет:

```
5 67
11
9
```

Напишем программу, которая считывает 2 числа и выводит на экран их сумму:

```
#include <iostream>

using namespace std;

int main() {
    int a, b;
    cin >> a >> b;
    cout << a + b;
}
```

1.8 Вычисления по модулю

В случае, когда ответ слишком большой и не помещается в тип `long long`, часто требуется вывести его по модулю $10^9 + 7$ или какому-либо другому. Это значит, что надо найти остаток от деления ответа на заданное число. Для решения таких задач полезно помнить следующее утверждение:

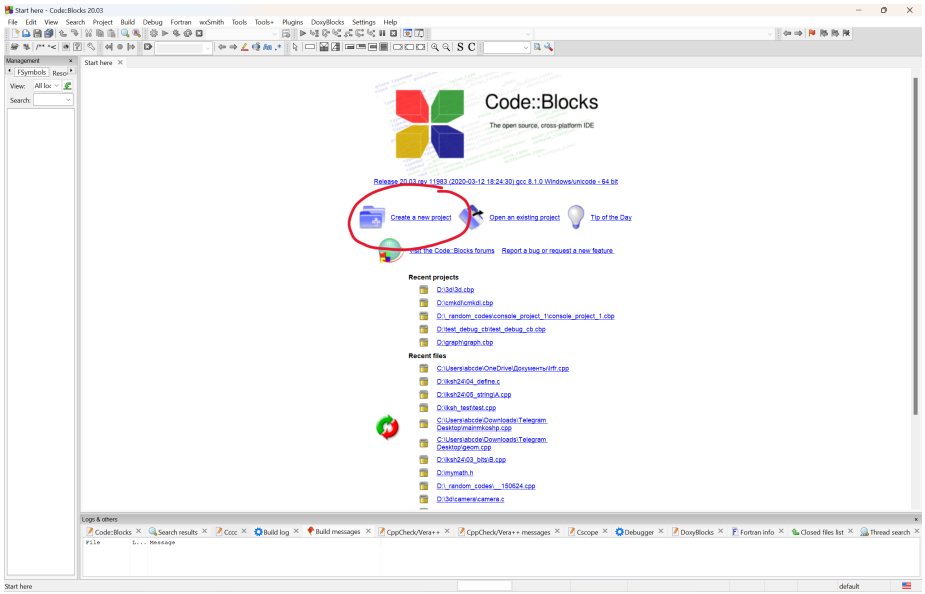
Утверждение. Остаток суммы (разности, произведения) при делении на m равен остатку суммы (разности, произведения) остатков некоторого количества чисел при делении на m.

1.9 Установка и использование Code::Blocks (Windows)

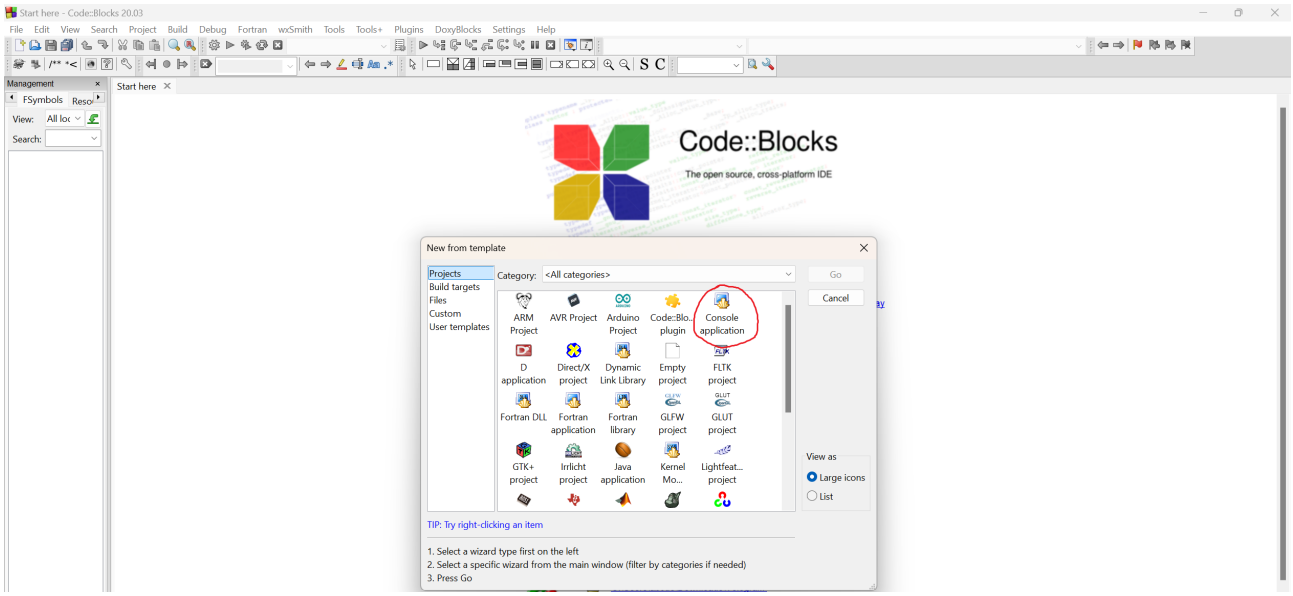
Мы будем работать со средой `Code::Blocks`. В случае, если вы хотите более красивую среду и у вас `Linux`, то перейдите в раздел 8.3, в котором описывается установка среды VS Code. В случае, если у вас Windows и вы хотите установить VS Code, можете написать в бота, мы постараемся вам помочь.

Чтобы установить Code::Blocks, перейдите на страницу https://www.codeblocks.org/downloads/binaries/. Выберите файл `codeblocks-20.03mingw-setup.exe`. Нажмите FossHUB рядом с ним. Установите файл и запустите его. Оставляйте все настройки по умолчанию. После этого запустите Code::Blocks.

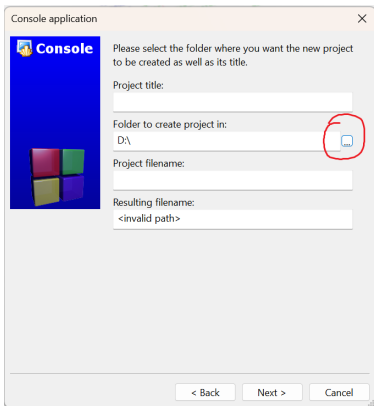
1. Работать будем в проектах. Чтобы создать проект, нажмите на кнопку `Create a new project`.



2. Выберите `Console application`.



3. Нажмите `Next`. Выберите C++, нажмите `Next`. Разберемся с следующим окном подробнее:

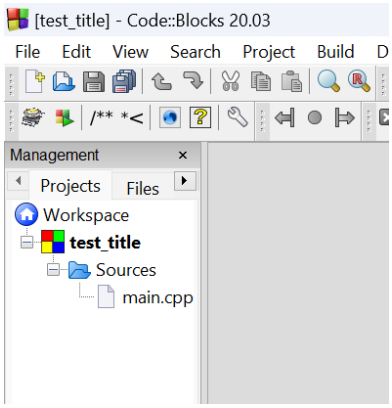


4. Укажите любое имя проекта (`Project title`) и папку, где проект будет создан (`Folder to create project in`). Для выбора папки нажмите на многоточие (обведено красным на картинке выше). Нажмите `Next`.

5. Нажмите `Finish`.

Проект создан. Теперь научимся открывать файл, в котором будем писать код.

1. Если у Вас открыта панель `Management` (слева), то откройте там вкладку `Projects`. Откройте все подпапки (нажав на плюс слева от названий). Нажмите дважды на `main.cpp`



2. Если панели `Management` нет, нажмите комбинацию клавиш `Shift + F2`.

Чтобы запустить код, нажмите клавишу `F9` или на значок  .

1.10 Структура олимпиадной задачи

Обычно олимпиадная задача состоит из нескольких частей. Разберем их все по отдельности:

1. **Ограничения**. Здесь указаны ограничения по времени и памяти. Если Ваша программа будет их превышать, вы получите вердикт Превышено ограничение времени (TL или TLE — от Time Limit Exceeded) или Превышено ограничения памяти (ML или MLE — от Memory Limit Exceeded).

2. **Имя входного/выходного файла**. Обычно здесь написано стандартный поток ввода (или `stdin`) и стандартный поток вывода (`stdout`). Если указано что-то иное, то входные данные надо читать из одного файла, а выходные данные выводить в другой файл, названия которых указаны в этой секции. О работе с файлами см. п. 8.4.

3. **Легенда**. Условия задач по программированию обычно содержат какую-либо интересную легенду. Например, людям дарят граф на Новый год. Но на начальном этапе, скорее всего, Вы будете встречаться с формальными условиями задач. В этой части описано общее содержание задачи.

4. **Входные данные**. В этой части описывается формат входных данных — того, что подаётся Вашей программе на вход. Также здесь описываются ограничения на размер входных данных, чтобы Вы могли подобрать алгоритм, который при таких ограничениях укладывается в отведенное время.

5. **Выходные данные**. В этой части описывается формат выходных данных — того, что должна выводить Ваша программа.

6. **Система оценки**. Здесь описывается оценка задачи. Подробнее про оценку задачи см. п. 1.11. При `ICPC`-формате эта секция отсутствует. В `IOI`-формате здесь приводятся ограничения на каждую из подгрупп или указывается на потестовую оценку.

7. **Примеры**. Здесь приводятся один или несколько примеров входных данных и соответствующих им выходных данных. Обратите внимание, что в тестирующей системе есть другие тесты, которые неизвестны Вам. Поэтому программа должна работать не только на тестах из условия, но и в общем случае.

8. **Замечание**. В этой секции объясняются примеры или даются иные комментарии по задаче. Часто эта секция отсутствует.

1.11 Оценка олимпиадной задачи

Есть два наиболее популярных вида оценки задач:

1. ICPC

   Каждая задача либо зачтена, либо не зачтена. Чтобы задача была зачтена (вердикт `OK/AC/Полное решение`, нужно, чтобы решение прошло все тесты). Скорее всего, большая часть школьных задач и задач на обучающих курсах будет именно в этом формате. Этот формат используется на Внутренней олимпиаде 54-ой школы (начиная с IV-ой) и на командных олимпиадах, как среди школьников, так и среди студентов.

   Штраф. Так как возможных результатов довольно мало (количество задач плюс 1), то участники, решившие одинаковое число задач, упорядочивается по штрафу. Штраф зависит от времени, прошедшего с начала соревнования до успешной сдачи задачи, и от количества неудачных посылок. В учебных задачах штраф не важен.

2. IOI

   В этом формате каждое решение оценивается некоторым количеством баллов (обычно от 0 до 100). Количество баллов может определяться по-разному. Ниже приведены два наиболее распространённых вида:

   (a) **Оценка по подгруппам**. Задача разбита на несколько подгрупп. На каждую подгруппу, кроме, скорее всего, последней, наложены дополнительные условия, упрощающие задачу. Обычно баллы за подгруппу ставятся тогда, когда пройдены все тесты этой подгруппы.

   (b) **Потестовая оценка**. В задаче n тестов, кроме тестов из условия, и каждый из них стоит $\frac{100}{n}$ баллов.

   Этот формат используется на всех этапах Всероссийской олимпиады школьников, Летнем кубке по программированию 54-ой школы, на многих перечневых олимпиадах и на Международной олимпиаде по информатике (`IOI`). I-III Внутренние олимпиады 54-ой школы также использовали этот формат.

1.12 Вердикты тестирующей системы

Ниже приведены наиболее часто встречающиеся вердикты:

Полное название	**Английское название**	**Сокращение**	**Причины**
Полное решение	OK/Accepted	OK/AC	Решение работает верно на всех тестах жюри
Неправильный ответ	Wrong answer	WA	Решение выводит неверный ответ
Превышено время исполнения	Time Limit (Exceeded)	TL/TLE	Решение работает слишком долго (из-за неподходящего алгоритма или ошибки в реализации)
Превышен лимит по памяти	Memory Limit (Exceeded)	ML/MLE	Решение выделяет слишком много памяти
Ошибка исполнения	Runtime Error	RE	Решение исполняется с ошибкой
Ошибка компиляции	Compilation Error	CE	Решение не компилируется
Ошибка представления	Presentation Error	PE	Формат вывода программы не соответствует выходным данным. В некоторых системах отображается как WA.

Далее приведены специфичные вердикты:

Полное название	Английское название	Сокращение	Причины
Нарушение стиля оформления программ	Style Violation	SV	Пробелы в коде стоят не там, где надо, отступы не соответствуют требуемым и т.д.
Ожидает подтверждения	Pending Review	PR	Посылка прошла тесты, но ожидает ручного подтверждения
Отклонено	Rejected	RJ	Преподаватель признал попытку неверной
Превышен лимит бездействия/Решение «зависло»	Idleness Limit (Exceeded)	IL/ILE	Этот вердикт предназначен для интерактивных задач (подробнее см. п. 8.5) и возникает, когда программа слишком долго не взаимодействует с интерактором. Обычно происходит, если в решении не сбрасывается буфер потока вывода.

1.13 Числа с плавающей точкой (Дробные числа)

До этого мы работали только с целыми числами. Теперь познакомимся с дробными числами. Из-за особенностей их хранения в памяти, вычисления с ними не точны, поэтому лучше использовать целые числа, если есть такая возможность.

Для хранения в памяти дробных чисел существуют типы `float`, `double`, `long double`. Каждый из них умеет хранить числа точнее, чем предыдущий. Не рекомендуется использовать тип `float` из-за его маленькой точности. Арифметические операции осуществляются с ними так же, как и с целыми. Но оператор деления делит не нацело, а «как в математике» $\left(\frac{5.0}{2.0} = 2.5\right)$. Для выполнения деления без округления хотя бы один из операндов должен быть числом с плавающей точкой. Для приведения целого числа к типу `double` его можно умножить на `1.0`, для приведения к типу `long double` — на `1.0l`. Суффикс `l` (латинская строчная L) показывает, что число имеет тип `long double`.

Обратите внимание, что десятичным разделителем является точка.

Еще одна возможная форма записи чисел с плавающей точкой — научная. Например, `2.4e6` означает то же, что и $2.4 \cdot 10^6$. Для записи больших целых чисел тоже используется такой формат. Например, можно часто встретить

```
const int INF = 2e9;
```

или

```
const int MOD = 1e9 + 7;
```

Но на самом деле, C++ создает дробное число, а затем преобразует его к целому.

Есть особенность, связанная с выводом чисел с плавающей точкой. Познакомимся с ней на примере. Запустим следующий код:

```
#include <iostream>

using namespace std;

int main() {
    long double a = 1001;
    cout << a * a * a << '\n';
}
```

Этот код считает 1001^3 в дробных числах. Мы ожидаем получить ответ 1003003001, но после запуска увидим $1.003e{+}09$. Ответ посчитан верно, но с очень низкой точностью. В задачах обычно указана точность ответа, но лучше всегда выводить максимальное возможное количество знаков после запятой, если вывод лишних знаков не противоречит условию. Чтобы управлять точностью выводимых чисел, допишите после `#include <iostream>` строку `#include <iomanip>`. В следующей строке после `int main() {` напишите `cout « fixed « setprecision(k);`, где вместо k укажите количество цифр, которое необходимо вывести. Этот код всегда будет выводить k цифр после запятой. Если не требуется выводить нули после запятой, можно опустить `fixed «`.

Тогда наш исправленный код будет выглядеть так:

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

```

```cpp
using namespace std;

int main() {
    cout << fixed << setprecision(100);
    long double a = 1001;
    cout << a * a * a << '\n';
}
```

1.14 Развеевание магии

Поговорим о назначении команд `#include`.

Команда `#include` *подключает библиотеки*, то есть позволяет использовать код, написанный другими разработчиками (в олимпиадном программировании — обычно авторами языка). Название библиотек указывается в треугольных кавычках. Приведём наиболее популярные из встроенных библиотек:

Название	**Описание**
`iostream`	Работа с вводом-выводом
`iomanip`	Настройка ввода-вывода (точность вывода дробных чисел и т.п.)
`cmath`	Расширенная математика (см. след. раздел)
`vector`	Векторы (динамические массивы) — будут рассмотрены позже
`algorithm`	Алгоритмы (сортировка, бинарный поиск). Вы познакомитесь с ней в 9 классе

Команда `using namespace std;` позволяет нам писать код короче. Например, задача «ввести 2 числа и найти их сумму» без неё решается так:

```cpp
#include <iostream>

int main() {
    int a, b;
    std::cin >> a >> b;
    std::cout << a + b << std::endl;
}
```

На написание такого кода требуется больше времени, но в промышленной разработке такой способ предпочтительнее, так как позволяет использовать одно и то же название несколько раз.

1.15 Расширенная математика

Мы научились выполнять арифметические операции. Но часто требуются и другие математические операции: корень, тригонометрические функции, логарифм. Чтобы использовать их, требуется подключить библиотеку `cmath`.

Ниже приведена таблица функций:

Название в C++	**Описание**
sqrt	Квадратный корень
cbrt	Кубический корень
pow	Возведение в степень
abs	Модуль
log	Натуральный логарифм
log10	Десятичный логарифм
log2	Двоичный логарифм
sin, cos, tan	Тригонометрические функции
asin, acos, atan	Обратные тригонометрические функции
floor, ceil, round	Округление (вниз, вверх, к ближайшему)
M_PI	Число Пи

Важно! Все тригонометрические функции работают в радианах (1 радиан $= \frac{\pi}{180}^\circ$).

Важно! Возведение в степень работает только в дробных числах, поэтому в случае целых чисел надёжнее реализовать руками.

2 Ветвление

2.1 Тип данных `bool`. Логические условия

На данный момент нам известно два типа данных — `int` и `long long`. Познакомимся ещё с одним типом данных — `bool`. Тип данных `bool` хранит одно из двух возможных значений: `Истина (да)` или `Ложь (нет)`. Например, выражение $5 > 4$ равносильно `Истине`, а $5 > 5$ равносильно `Лжи`. В C++ `Истина` обозначается `true`, а `Ложь` — `false`. Вы можете создать переменную типа `bool` так же, как и переменную других известных Вам типов:

```
bool a;
```

или

```
bool a = true;
```

Также можно выводить переменную типа `bool`. При выводе значения `true` будет напечатана единица, при выводе `false` — ноль. При вводе `bool` происходит считывание числа. Если оно равно 0, то результатом будет `false`; иначе результатом будет `true`.

В C++ есть возможность сравнивать числа с помощью операторов $>$, $<$, $\geq$, $\leq$, $=$, $\neq$, но форма их записи отличается. В таблице ниже приведено обозначение этих операторов в C++:

Математика	**C++**
$>$	`>`
$<$	`<`
$\geq$	`>=`
$\leq$	`<=`
$=$	`==`
$\neq$	`!=`

Результат сравнения имеет тип `bool`. Например, после выполнения кода

```
bool a = 5 > 3;
```

в переменной `a` будет храниться значение `true`.

2.2 Операторы над типом `bool`

Тип `bool` кажется довольно странным, но позже мы увидим, что он часто оказывается полезен. Пока познакомимся с операторами для работы с этим типом. В таблице ниже приведены названия и обозначения в C++ каждого из операторов.

Оператор в C++	**Название**
&&	логическое И
||	логическое ИЛИ
==	равенство
!=	неравенство

Последние два оператора работают так же, как и с остальными типами данных. Остановимся на первых двух подробнее.

Результат логического И для двух операндов истинен тогда и только тогда, когда оба операнда истинны. Таблица истинности для этого оператора:

A	B	$A \&\& B$
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Результат логического ИЛИ истинен тогда и только тогда, когда хотя бы один из двух операндов истинен. Таблица истинности для него:

A	B	$A \|\| B$
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Кроме того, для типа `bool` определён унарный оператор отрицания, обозначаемый `!`. Он меняет значение, к которому применяется, на противоположное. Таким образом, его таблица истинности выглядит так:

A	$!A$
`false`	`true`
`true`	`false`

Стилистическая заметка. Напоминаем, что бинарные операторы (в т.ч. логические) отделяются от своих операндов пробелами. Между унарным оператором и его операндом пробел не ставится.

Замечание. Если логическим операторам передавать числа как операнды, то они будут преобразованы к типу `bool` по следующему правилу: 0 преобразуется в `false`, все остальные числа — в `true`.

2.3 Неполное ветвление. Ключевое слово `if`

Мы научились сравнивать числа. Но сейчас наша программа выполняется строчка за строчкой, мы пока не можем выполнять различные действия в зависимости от некоторых условий. Но такая возможность есть, и в этом пункте мы познакомимся с ней. Фраза «Если выполнено условие A, то сделай B» на C++ записывается так:

```
if (A) {
    B;
}
```

После выполнения этого блока кода, будет выполнена следующая после закрывающей фигурной скобки строчка.
«Место B» называется *телом* условной конструкции, A — *условием*.
Тело может состоять из нескольких инструкций.
Напишем код, который выполняет следующий следующий алгоритм:

1. Считать число.
2. Если число больше 5, вычесть из него 1.
3. Вывести число.

```
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;
    if (x > 5) {
        x = x - 1;
    }
    cout << x << '\n';
}
```

Стилистическая заметка. Отступ в теле условной конструкции должен быть на один tab (4 пробела) больше, чем отступ вне её.

Условие может состоять из нескольких логических выражений, объединённых логическими операторами, которые мы изучили ранее. Решим следующую задачу:

Задача. Вводится целое число x. Если $5 \leq x \leq 10$, то выведите 1, иначе не выводите ничего.

```
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;
    if (x >= 5 && x <= 10) {
        cout << 1;
    }
}
```

Важно! Запись `5 <= x <= 10` в C++ не равносильна математическому двойному неравенству $5 \leq x \leq 10$, в отличие от других языков (например, Python), которые могут допускать такую запись. Корректно писать так, как в приведённом выше примере.

2.3.1 Область видимости переменной

Важно ! Переменная, объявленная внутри фигурных скобок доступна только внутри них. Например, следующий код **содержит ошибку**, так как переменная `b` была объявлена внутри `if` и доступна только в его теле. Но в теле условной конструкции можно получать доступ к переменной `a`, так как тело условной конструкции находится внутри фигурных скобок `main`.

```
#include<iostream>

using namespace std;

int main() {
    int a = 5;
    if (a > 3) {
        int b = 10;
    }
    cout << b;
}
```

Тот участок кода, где переменная доступна, называется *областью видимости* этой переменной.

2.4 Полное ветвление. Ключевое слово `else`

Ключевое слово `else` может идти только после условной конструкции. Синтаксис при его использовании таков:

```
if (A) {
    B;
} else {
    C;
}
```

Блоки `A` и `B` были разобраны в прошлом пункте. Инструкции блока `C` выполняются только в том случае, когда не выполнилось условие `A`. В блоке `C`, как и в блоке `B`, может быть несколько инструкций. Например, в прошлом пункте мы решали задачу о принадлежности числа отрезку $[5; 10]$. Если число не принадлежало отрезку, то мы ничего не выводили. Давайте в таком случае выведем 0:

```
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;
    if (x >= 5 && x <= 10) {
        cout << 1;
    } else {
        cout << 0;
    }
}
```

2.5 Использование `else-if`

Мы научились делать что-либо, если какое-то условие выполнено, и научились выбирать из двух вариантов действий нужный, исходя из истинности условия. Теперь рассмотрим, что делать, если вариантов действий больше двух. Например, решим такую задачу:

Задача. Требуется определить возрастную категорию человека по его возрасту в годах x. Если человеку ещё нет 7 лет, отнесём его к категории 0 (дошкольники), если уже есть 7, но ещё нет 18, — к категории 1 (школьники), если есть 18 — к категории 2 (взрослые).

```
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;
    if (x < 7) {
        cout << 0;
    } else if (x < 18) {
        cout << 1;
    } else {
        cout << 2;
```

```
    }
}
```

Когда C++ видит такую конструкцию, он последовательно проверяет условия. Когда он находит истинное, он выполняет тело после этого условие, а затем пропускает все остальные. Если ни одно из условий не выполнилось, выполняется тело `else`. Именно поэтому во втором условии нет проверки $x \geq 7$, так как, если это условие было бы не выполнено, то выполнился бы первый блок.

Конструкций `else-if` может быть много. Например,

```
#include <iostream>

using namespace std;

int main() {
    int x;
    cin >> x;
    if (x < 7) {
        cout << 0;
    } else if (x < 18) {
        cout << 1;
    } else if (x < 35) {
        cout << 2;
    } else {
        cout << 3;
    }
}
```

2.6 Вывод текста на экран

Часто в задачах, где надо проверить какие-либо условия, требуется вывести «YES» или «NO». Чтобы вывести текст, надо записать его в кавычках ("), чтобы C++ не думал, что слова в нём — названия переменных. Например, чтобы вывести «YES» (без кавычек), нужно написать:

```
cout << "YES";
```

3 Циклы. Цикл с предусловием. Цикл «n раз». Цикл с постусловием. Вложенные циклы.

Наш код уже может выполняться по-разному в зависимости от разных условий. Теперь рассмотрим способы «зациклить» наш алгоритм — повторить его некоторое количество раз. Перед этим познакомимся с составными арифметическими операторами, которые зачастую удобны при написании циклов.

3.1 Составные арифметические операции

Мы уже знаем, что для увеличения переменной x на 1 можно написать следующую инструкцию:

```
x = x + 1;
```

Увеличение переменной на 1 часто используется в циклах, поэтому для этой операции есть сокращённая запись:

```
++x;
```

`++` — это оператор *инкремента*. Точнее, приведённая выше запись — это *префиксный инкремент*. Также существует *постфиксный инкремент*:

```
x++;
```

Они оба увеличивают значение переменной на 1.

Замечание . Разница между префиксным и постфиксным инкрементом проявляется, если результат выполнения присвоить другой переменной. При использовании префиксного инкремента, значение новой переменной будет равно новому значению инкрементируемой, а при использовании постфиксного инкремента — старому. Кроме того, префиксный инкремент работает быстрее постфиксного, хотя и несильно.

Существуют и другие составные операторы. Они приведены в таблице ниже:

Сокращённая запись	Полная запись
`a--;`	`a = a - 1`
`a += b;`	`a = a + b;`
`a -= b;`	`a = a - b;`
`a *= b;`	`a = a * b;`
`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`

3.2 Цикл с предусловием (`while`)

Рассмотрим самый простой из циклов — цикл с предусловием (цикл `while` — от англ. «пока»). Он имеет такой же синтаксис, как и `if` (в круглых скобках условие A, затем в фигурных скобках тело цикла B). Когда C++ выполняет этот блок кода, он поступает так:

1. Если A выполнено, сделать B.
2. Если A выполнено, сделать B.
3. Если A выполнено, сделать B.

   ...

Как только условие перестаёт быть выполнено, C++ переходит к коду после цикла.
Следующий код выводит числа от 1 до 5:

```
int i = 1;
while (i <= 5) {
    cout << i << endl;
    i++;
}
```

Стилистическая заметка. Тело циклов (как этого, так и остальных) пишется с отступом на 4 пробела больше, чем у кода вне цикла.

Стилистическая заметка. Наиболее распространённые названия переменных для счётчиков циклов — i, j, k. От них ожидается такое предназначение, в отличие от a, x, m.

Один проход по телу цикла называется *итерацией*.

3.3 Цикл «n раз» (`for`)

Рассмотрим тот же код, что и в прошлом примере. Оказывается, такие конструкции — перебор чисел в каком-то диапазоне с фиксированным шагом — встречаются довольно часто. Но при написании такой конструкции легко ошибиться, например, пропустить `i++`. Поэтому в языке C++ существует другой цикл, упрощающий такую конструкцию. Этот цикл называется *циклом `for`*. Можно предположить, что это связано с его языковым аналогом «для каждого числа от ... до ... с шагом ... выполнить ...». Познакомимся с его синтаксисом:

```
for (A; B; C) {
    D;
}
```

A содержит инструкцию, выполняемую до цикла один раз. Чаще всего, это инициализация счётчика.
B содержит условие, которое проверяется перед каждой итерацией. Если условие не выполняется, цикл завершается.
C содержит инструкцию, выполняемую после каждой итерации. Обычно это изменение счётчиков.
D — тело цикла, то есть то, что выполняется на каждой итерации.
Рассмотрим задачу:
Задача. Вводится число n ($1 \le n \le 10^5$). Выведите все натуральные числа от 1 до n (включительно).
Мы уже умеем решать эту задачу циклом `while`. Теперь решим её циклом `for`:

```
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cout << i << endl;
    }
}
```

Перед циклом мы создаём переменную i, изначально равную единице. Условие выполнения нашего цикла — $i \leq n$. После каждой итерации мы увеличиваем i на 1. На каждой итерации мы выводим i.

Замечание. Цикл `for` может быть заменён на `while` (выше текстом описано, как это сделать). Но это не является хорошим тоном, так как понижает читабельность кода.

Замечание. Некоторые части могут опускаться, например, ниже приведён корректный код:

```
int i = 1;
for (; i <= 5; i++) {
    cout << i << endl;
}
```

Замечание. `for(;;)` — бесконечный цикл, выполняющийся неограниченное число раз.

3.4 Цикл с постусловием (`do-while`)

Циклы `while` и `for` проверяют условие до выполнения цикла. Из этого следует, что если условие изначально ложно, то цикл не будет выполнен ни разу. Иногда это неудобно. Для этого создан цикл с постусловием. В отличие от цикла `while`, он выполняет следующее:

1. Выполнить B.
2. Если A, выполнить B.
3. Если A, выполнить B.
4. Если A, выполнить B.

   ...

Таким образом, тело цикла `do-while` всегда выполняется хотя бы один раз.

Познакомимся с синтаксисом этого цикла в C++:

```
do {
    B;
} while (A);
```

Условие проверяется после выполнения тела и записывается после него. Обратите внимание на `;` после условия.

4 Массивы

4.1 Одномерные статические массивы

Часто в задачах возникает необходимость хранить некоторую последовательность значений довольно большой или заранее неизвестной длины. В таком случае применяются *массивы*. Массив — это упорядоченный набор значений одного типа. В этой главе мы познакомимся со статическими массивами. Динамические массивы будут разобраны позже.

Замечание. Статические массивы — «наследие» языка Си, предшественника C++.

Чтобы создать массив используется следующий синтаксис:

```
type name[size];
```

`type` — это тип значений в массиве, `name` — название переменной (имя массива), `size` — размер массива, то есть количество значений, которое будет в нём храниться.

Кроме того, элементам массива можно сразу присвоить значения:

```
int arr[5] = {1, 2, 3, 4, 5};
```

Обратите внимание, что элементы массива перечисляются в фигурных скобках, в отличие от многих других языков программирования (Python, JavaScript).

Чтобы заполнить массив нулями, можно использовать такой синтаксис:

```
int arr[5] = {0};
```

Важно! Чтобы заполнить массив числами, отличными от нуля, нужно использовать цикл `for`.

Чтобы обращаться к элементам массива, используются квадратные скобки. Например, `arr[1]`.

Важно! Индексация (то есть нумерация элементов) массива начинается с **нуля**, а не с единицы. Поэтому если использовать приведённый выше массив, то `arr[0]` будет равно 1, а `arr[1]` — 2.

Массивы часто используется вместе с циклами. Например решим следующую задачу:

Задача. Дано число n, на следующей строке последовательность из n чисел через пробел. Выведите эту последовательность в обратном порядке.

```cpp
#include <iostream>

using namespace std;

int main() {
    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    for (int i = n - 1; i >= 0; i--) {
        cout << a[i] << ' ';
    }
}
```

Замечание. При работе со статическими массивами, их размер, согласно стандарту, требуется делать известным заранее (например, 10^5). Несмотря на это, компилятор GCC допускает использование массивов так, как в приведённом выше примере.

Написанный по стандарту код будет выглядеть так:

```cpp
#include <iostream>

using namespace std;

int main() {
    const int ARRAY_SIZE = 1e5;
    int n;
    cin >> n;
    int a[ARRAY_SIZE];
    for (int i = 0; i < n; i++) {
        cin >> a[i];
    }
    for (int i = n - 1; i >= 0; i--) {
        cout << a[i] << ' ';
    }
}
```

Стилистическая заметка. В случае использования констант, их следует объявлять как константы (так, как в приведённом выше примере).

Замечание. Выделение лишней памяти негативно сказывается на производительности. Но в задачах это не существенно, так как время и память оцениваются по наибольшим показателям среди всех тестов, а среди тестов обязательно встретится тест с максимальными ограничениями. Позже мы познакомимся с динамическим массивом (`vector`), который является более гибким.

4.2 Двумерные массивы

Двумерные массивы служат для представления таблиц. Чтобы создать массив размера $n \times m$ требуется использовать `type name[n][m]` — аналогично одномерным.

Решим следующую задачу:

Задача. Даны числа n и m и массив целых чисел размера $n \times m$. Посчитайте сумму в каждой строке и в каждом столбце.

```cpp
#include <iostream>

using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    int arr[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> arr[i][j];
        }
    }
    int sum_row[n];
    for (int i = 0; i < n; i++) {
        sum_row[i] = 0;
        for (int j = 0; j < m; j++) {
            sum_row[i] += arr[i][j];
```

```
        }
    }
    int sum_col[m];
    for (int i = 0; i < m; i++) {
        sum_col[i] = 0;
        for (int j = 0; j < n; j++) {
            sum_col[i] += arr[j][i];
        }
    }
    for (int i = 0; i < n; i++) {
        cout << sum_row[i] << ' ';
    }
    cout << '\n';
    for (int i = 0; i < m; i++) {
        cout << sum_col[i] << ' ';
    }
    cout << '\n';
}
```

5 Символы. Строки.

5.1 Символы

Раньше мы работали только с числами и массивами чисел. Теперь научимся работать с текстом. Познакомимся с новым типом данных C++ — символ (`char`). По умолчанию C++ работает с символами по таблице ASCII (American Standard Code for Information Interchange) (читается [аски]), которые содержат только латинские буквы, цифры, знаки пунктуации и некоторые служебные символы (перевод строки, возврат каретки и т.д.). В таблице ASCII каждому символу сопоставлен номер от 0 до 127. Таким образом, номер символа можно хранить в 7 битах, но, из-за особенностей устройства памяти в компьютере, тип `char` занимает 1 байт (8 бит). Ниже приведена таблица ASCII:

32	пробел	48	0	64	@	80	P	96	‘	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	|	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		

Символы до №31 все являются служебными.

Значения этого типа пишутся в одинарных кавычках, например,

```
char c = 'x';
```

Выведем *c* на экран. На экране мы увидим символ `x`. Теперь рассмотрим следующий код:

```
char c = 'x';
int a = c;
cout << a;
```

На экране мы увидим число 120 — номер символа `x`.

5.1.1 Арифметические операции с символами

Благодаря тому, что символам сопоставлены числа, мы можем работать с ними, как с числами. Рассмотрим следующий пример:

Задача. Дан символ, являющийся строчной буквой латинского алфавита. Определить его номер в алфавите (считая, что букве `a` соответствует номер 1).

Вычтем из данного нам символа символ `a`. Тогда мы узнаем номер символа в 0-индексации. Добавим 1, чтобы получить его в 1-индексации.

```
char c;
cin >> c;
cout << c - 'a' + 1;
```

Другой пример:

Задача. Дано число x. Выведите x-ю строчную букву латинского алфавита.

Теперь нам нужно решить задачу, обратную предыдущей. Напишем такой код:

```
int x;
cin >> x;
cout << 'a' + (x - 1);
```

Запустим его. Введём, например, 5. На экране мы увидим 101. Это связано с тем, что результат арифметических действий над символами вычисляется в типе `int`. Чтобы исправить это, изменим код так:

```
int x;
cin >> x;
cout << (char)('a' + (x - 1));
```

Теперь код, как и должно быть, выводит `e`. Мы явно указали C++, что требуется *привести* тип `int` к типу `char`.

Упражнение. Вводится символ, являющийся цифрой. Считайте его как символ, а затем преобразуйте к соответствующему числу (в тип `int`).

5.1.2 Логические операторы

С символами определены и операторы сравнения (>, < и т.д.). Символы сравниваются в порядке номеров в таблице ASCII. Наиболее часто пригождается проверка, является ли символ буквой (строчной буквой, заглавной буквой, цифрой и т.д.).

Так как строчные буквы (как и заглавные буквы и цифры) идут в таблице ASCII последовательно, для этого достаточно сравнить символы. Например, так:

```
char c;
cin >> c;
if ('A' <= c && c <= 'Z') {
    cout << "YES";
} else {
    cout << "NO";
}
```

5.2 Строки

Строкой в программировании называется любая последовательность символов. Для хранения строк используется тип `string`. Для него определён ввод и вывод (через `>>` и `<<` соответственно). Для обращения к номерам символов используется такой же синтаксис, как и у массивов, — квадратные скобки.

Важно ! Нумерация символов в строке начинается с 0.

Кроме того, для скобок определен оператор `+`, который выполняет *конкатенацию* строк, то есть их «склеивание». Кроме того, для строк определён *метод* `substr`, который позволяет получить *подстроку* (то есть непрерывный участок исходной строки).

Первое число указывает индекс начала строки, второе число — длину подстроки. Если второе число не указано, то конец подстроки совпадает с концом строки. Познакомимся с методом `substr` на примере:

```
string s = "abcdef";
s.substr(1, 3); // "bcd"
s.substr(2); // "cdef"
```

Строки можно вводить аналогично числам (`cin >> string_name`), но этот способ читает строку до пробела, не включая пробел. При этом пробел будет пропущен, то есть не будет получен при следующем чтении. В случае, если нужно считать строку целиком, можно использовать функцию `getline`. Её синтаксис таков:

```
string string_name;
getline(cin, string_name);
```

Есть и другой способ считывания строки с пробелами: выполнять считывание, пока не закончится файл входных данных (для тестирования у себя на компьютере используйте `Ctrl+D`, затем `Enter` для завершения ввода). Стандартная реализация выглядит так:

```
string word;
while (cin >> word) {

}
```

В теле цикла обрабатывают очередное слово.

Так можно поступать и с последовательностями чисел неизвестной длины.

Отметим, что существуют функции `to_string` и `stoi` (от `string to int`), которые преобразуют целое число в строку и наоборот.

6 Процедуры. Функции. Рекурсия.

Циклы помогли нам избежать повторов аналогичного кода. Отсутствие повторов — это хорошо, потому что:

1. Код меньше.
2. Если была допущена ошибка, её надо исправить только в одном месте.

Функции — это ещё один способ избавиться от повторов.

6.1 Процедуры

Разберём несколько задач, чтобы понять, что такое процедура.

Задача. Три раза повторите следующее: считайте строку s и выведите «`Hello, `», а затем строку s.

Выделим повторяющуюся часть кода. Здесь это чтение строки s и вывод приветствия. Выделим это в отдельный «блок кода», назовём его `hello`:

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

using namespace std;

void hello() {
    string s;
    cin >> s;
    cout << "Hello, " << s << '\n';
}

int main() {
    for (int i = 0; i < 3; i++) {
        hello();
    }
}
```

Мы определили процедуру `hello`. Внутри фигурных скобок написано то, что она делает. Чтобы *вызвать* (выполнить) её, нужно написать её имя (в данном случае, `hello`), а затем круглые скобки, что мы и делаем в цикле.

Задача. Дано число n. Вывести n строк, в i-й строке вывести i символов «*».

Попробуем найти в задаче повторяющийся алгоритм. В данном случае, это вывод некоторого количества звёздочек в строку, но их количество отличается. Теперь у алгоритма появился *параметр* — количество звёздочек. Поэтому мы можем создать процедуру, которая будет *принимать* параметр x и печатать x звёздочек. В основном коде мы n раз *вызовем* (выполним) эту процедуру для каждого x от 1 до n. В коде это будет выглядеть так:

```
#include <iostream>

using namespace std;

void printStars(int x) {
    for (int i = 0; i < x; i++) {
        cout << '*';
    }
    cout << '\n';
}

int main() {
    int n;
    cin >> n;
    for (int i = 1; i <= n; i++) {
        printStars(i);
    }
}
```

Для каждого параметра указывается его тип и название, параметры перечисляются через запятую.

6.2 Функции

Процедуры умеют совершать аналогичные действия для разных параметров. Но часто требуется, чтобы функция, как в математике, «выдавала» (*возвращала*) некоторый результат.

Функции определяются так же, как и процедуры, но вместо `void` указывается тип того значения, которое возвращает функция. Процедуры обычно тоже считаются функциями.

Пусть нам нужна функция, которая вычисляет длину гипотенузы по двум катетам. Тогда она принимает два вещественных числа, длины катетов, и возвращает одно вещественное число — длину гипотенузы. Эта функция будет выглядеть так:

```
long double hyp(long double a, long double b) {
    return sqrt(a * a + b * b);
}
```

Обратите внимание на ключевое слово `return`. После него следует то значение, которое возвращает функция.

Замечание. Функция, аналогичная нашей `hyp`, определена в библиотеке `cmath` и называется `hypot`.

Кроме того, следует заметить, что `sqrt`, `abs` и т.п. — тоже функции, но они определены не нами, а авторами библиотеки `cmath`.

Замечание. Безусловно, примеры, приведённые в этой главе довольно надуманы. Несмотря на это, функции являются очень важными при написании крупного кода. Кроме того, скоро мы познакомимся с рекурсией, использование которой часто помогает написать более простой код.

Замечание. Названия функций должно удовлетворять тем же требованиям, что и имена переменных.

Стилистическая заметка. Функциям следует давать имена, соответствующие их назначению.

6.3 Снова об области видимости. Глобальные переменные

В пункте 2.3.1 мы говорили об области видимости переменных. В частности, переменные, объявленные в функции доступны только в ней. Чтобы переменные были доступны во всех функциях, их надо объявить в *глобальной области*, например, так:

```
#include<iostream>

using namespace std;

int a;

int func() {
    cout << a * a;
}

int main() {
    cin >> a;
    func();
}
```

Стилистическая заметка. В промышленной разработке использование глобальных переменных является плохим стилем, но в олимпиадном программирование они скорее приветствуются.

6.4 Передача массива в функцию

Чтобы передать статический массив в функцию, надо указать звездочку после типа, который хранится в этом массиве. Обычно следующим параметром передаётся длина массива. Например, напишем функцию, вычисляющую сумму чисел в массиве:

```
int countSum(int* a, int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += a[i];
    }
    return sum;
}
```

Следует отметить, что в примере выше `a` — это *указатель* на тип `int`, но указатели выходят за рамки программы 8 класса.

6.5 Передача значений по ссылке

Проверим, копирует ли C++ значения при передаче их в функцию, то есть меняется ли значение переменных, переданных в функцию, если изменить их в функции. Для этого напишем следующий код:

```cpp
#include <iostream>

using namespace std;

void func(int x) {
    x++;
    cout << x << '\n'; // 6
}

int main() {
    int x = 5;
    func(x);
    cout << x << '\n'; // 5
}
```

Таким образом, C++ создаёт копии элементов при их передаче в функцию. Очень важно помнить об этом по двум причинам:

1. Иногда хочется, чтобы C++ не создавал копии, а передавал именно те же объекты. О том, как это сделать мы поговорим ниже.
2. В случае передачи «тяжёлых» объектов (например, векторов, с которыми мы познакомимся позже) функция может работать быстрее, чем копирование массива, которое бывает необязательным. Всегда передавайте векторы по ссылке, если это возможно.

Чтобы передать в функцию объект без копирования, после типа данных надо поставить амперсанд (`&`), например:

```cpp
#include <iostream>

using namespace std;

void func(int& x) {
    x++;
    cout << x << '\n'; // 6
}

int main() {
    int x = 5;
    func(x);
    cout << x << '\n'; // 6
}
```

6.6 О `main`

Теперь посмотрим на `main`. По синтаксису это функция, которая возвращает целое число, но ничего не принимает. Обратите внимание, что нами эта функция нигде явно не вызывается. Но почему же она возвращает целое число? И почему мы не используем `return`?

На самом деле, функция `main` возвращает то значение, которое возвращает программа. Обычно мы нигде не сталкиваемся с возвращаемым значением программ. Оно используется, чтобы указать, что программа завершилась без ошибок или с какой ошибкой завершилась программа. Если программа завершилась без ошибок, то возвращаемое значение должно быть равно 0. Иначе оно должно быть отлично от 0.

Большинство наиболее популярных компиляторов сами добавляют `return 0;` в конец функции `main`.

Важно! Если функция `main` будет возвращать значение, отличное от 0, программа в тестирующей системе получит вердикт `Ошибка исполнения`.

6.7 Рекурсия

Давайте попробуем вычислить факториал числа ($n! = 1 \cdot 2 \cdot \ldots n$, $0! = 1$). Мы можем написать решение, использующее циклы, но теперь попробуем написать другое решение.

Заметим, что $n! = n \cdot (n-1)!$. Тогда мы можем написать такой код:

```cpp
int fact(int n) {
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

Мы пользуемся приведённом выше свойством, чтобы вычислить значение функции-факториала *рекурсивно*: функция вызывает саму себя.

При написании рекурсивных функций важно помнить о *базовом случае* — условии, при котором не происходит рекурсивного вызова, так как, если бы его не было, рекурсия была бесконечной. В нашей функции базовый случай — $n = 0$.

Теперь попробуем научиться вычислять n-й член последовательности Фибоначчи, которая задаётся рекуррентно.

$$F_0 = 0, F_1 = 1, F_n = F_{n-2} + F_{n-1} \ (n \geq 2)$$

Реализация приведена ниже:

```
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

Теперь у нас 2 базовых случая ($n = 0$ и $n = 1$). Так тоже бывает.

Замечание . Данный код крайне не эффективен, так как мы многократно вычисляем значение для одного и того же числа. Для оптимизации можно применить технику *мемоизации* — запоминать уже вычисленные значения. Например, если мы знаем, что значение n не будет превосходить 50, то можно использовать следующий код:

```
const int N = 51;
int mem[N];

int fib(int n) {
    if (mem[n] != -1) return mem[n];
    if (n == 1 || n == 2) {
        return mem[n] = 1;
    }
    return mem[n] = fib(n - 1) + fib(n - 2);
}

int main() {
    for (int i = 0; i < N; i++) {
        mem[i] = -1;
    }
    int n;
    cin >> n;
    cout << fib(n) << '\n';
}
```

В приведённом выше коде мы запоминаем уже вычисленные значения в массиве `mem`, используя -1 как метку, что значение ещё не было вычислено. Массив имеет длину на 1 большую, чем максимальное число, которое может быть передано в функцию, так как индексация массива начинается с 0.

Обратите внимание, что мы можем возвращать результат присваивания. В таком случае, будет возвращено присвоенное значение.

7 Структуры данных

Мы уже знакомы со статическими массивами. Теперь изучим векторы и пары — важные структуры данных, встроенные в C++. После них мы познакомимся со структурами, которые позволяют объединять в один тип любые комбинации других типов, определять для них операторы и т.д.

7.1 Векторы (динамические массивы)

Ранее мы уже познакомились со статическими массивами, длина которых задана заранее и не изменяется. Теперь познакомимся с векторами.

Чтобы работать с векторами, требуется подключить библиотеку `vector`. Чтобы создать вектор, нужно написать

`vector<Type> name;`,

где `Type` — тип содержимого (например, `bool`), а `name` — имя переменной-вектора. Вектору можно сразу присвоить значение, например,

```
vector<int> a = {1, 2, 3};
```

Другой способ создать вектор — указать количество элементов и их значение. Приведённая ниже строка создаёт вектор из пяти элементов, каждый из которых равен -1.

```
vector<int> a(5, -1);
```

Можно указать только количество элементов, тогда вектор будет заполнен «стандартными» значениями данного типа (0 для чисел, пустая строка для строк и т.д.):

```
vector<int> a(5);
```

Важно! Длина вектора указывается в круглых скобках. Если вы попробуете использовать квадратные скобки, то вы создадите статический массив, элементы которого — векторы.

Обращение к элементам вектора по индексу осуществляется через квадратные скобки, так же, как и со статическими массивами. Индексация тоже начинается с нуля.

Теперь поговорим о возможностях вектора, которых нет у статических массивов. Самая примечательная особенность — возможность добавлять в конец или удалять последний элемент. Для этого используются *методы* `push_back` и `pop_back`.

Метод — это функция, относящаяся к определённому объекту. Чтобы вызвать метод, нужно написать имя объекта (в данном случае — имя переменной-вектора), через точку название метода, затем в круглых скобках через запятую аргументы.

Познакомимся с методами и их синтаксисом на примере:

```
vector<int> a = {1, 2, 3};
a.push_back(4); // a = {1, 2, 3, 4}
a.pop_back(); // a = {1, 2, 3}
a.pop_back(); // a = {1, 2}
```

Так как размер вектора может меняться, то мы можем узнать текущий размер массива. Для этого используется метод `size`:

```
vector<int> a = {1, 2, 3};
a.pop_back();
a.size(); // 2
```

Ещё одна особенность вектора — возможность перебирать его не только с помощью перебора индекса, но и *с помощью `for-auto`*. Например, приведённый ниже код выводит `54 5 4`.

```
vector<int> a = {54, 5, 4};
for (auto& i : a) {
    cout << i << ' ';
}
```

Часто такой перебор вектора используется для считывания массива. Например,

```
int n;
cin >> n;
vector<int> a(n);
for (auto& i : a) {
    cin >> i;
}
```

Этот код считывает сначала длину массива, а затем все его элементы.

Кроме возможности вставки в конец (`push_back`) и удаления из конца (`pop_back`), вектор может осуществлять вставку «в середину» (в произвольное место в массиве) и удаление «из середины». Рассмотрим пример. Возьмем массив `{1, 2, 3}`. Совершим с ним следующие операции:

1. Вставим элемент 4 по индексу 2. ←`{1, 2, 4, 3}`.
2. Удалим элемент по индексу 1. ←`{1, 4, 3}`

```
vector<int> a = {1, 2, 3};
a.insert(a.begin() + 2, 4);
a.erase(a.begin() + 1);
```

Методы `insert` и `erase` работают с *итераторами*, т.е. с особым типом данных, который указывает на определённое место в памяти. `begin` — итератор, указывающий на начало. Прибавив к нему i, мы получим итератор, указывающий на i-й элемент вектора, т.к. элементы вектора хранятся в памяти последовательно.

Важно! Операции `push_back` и `pop_back` осуществляют количество операций порядка 1, а `insert` и `erase` — порядка длины вектора, т.е. выполняются существенно дольше.

7.2 Двумерные векторы

В векторах можно хранить другие векторы. Например,

```
vector<vector<int>> a = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Обратите внимание, что для создания двумерного вектора заданного размера ($n \times m$) правильно писать так:

```
vector<vector<int>> a(n, vector<int>(m));
vector<vector<int>> b(n, vector<int>(m, 54));
```

Двумерный вектор `b` имеет размер $n \times m$ и заполнен числами 54.

7.3 Пара (`pair`)

Пара используется для хранения в одной переменной двух значений разных типов. Чтобы создать пару, используется следующий синтаксис:

```
pair<Type1, Type2> name;
```

Например,

```
pair<int, string> a = {54, "School No. 54"};
```

В паре можно хранить и одинаковые значения. В том числе пары часто хранятся в векторах вместо использования двумерных массивов, одна из размерностей которого — 2.

Чтобы обратиться к элементам пары, используются *свойства* `first` *и* `second`. Обращение к свойствам, как и к методам, осуществляется через точку, но после названия свойства не ставятся скобки.

7.4 Структуры (`struct`)

Чтобы объявить структуру, используется следующий синтаксис:

```
struct TypeName {
    Type1 name1;
    Type2 name2;
    ...
    TypeN nameN;
};
```

Важно! После объявления структуры ставится точка с запятой.

Например, пусть нам надо хранить информацию о клетке, у которой есть две целых координаты (`x` и `y`) и прочность (`durability`) — дробное число.

```
struct Cell {
    int x, y;
    double durability;
};
```

Тогда, чтобы создать переменную, хранящую информацию о клетке с координатами $(54, -54)$ и прочностью 5.4, используется следующий синтаксис:

```
Cell x = {54, -54, 5.4};
```

Поля в фигурных скобках перечисляются в том же порядке, что и при объявлении структуры.

Чтобы обратиться к свойству структуры, имя свойства пишут через точку после имени переменной. Например,

```
x.durability; // 5.4
x.x; // 54
x.y; // -54
```

7.4.1 Переопределение операторов

Рассмотрим следующую задачу:

Задача. Реализуйте структуру `Point`, содержащую две целых координаты. Реализуйте возможность сложения двух точек (оператором `+`). У точки-суммы каждая координата равна сумме соответствующих координат точек-операндов.

Решение приведено ниже:

```cpp
struct Point {
    int x, y;
};

Point operator+(Point a, Point b) {
    return {a.x + b.x, a.y + b.y};
}
```

Таким образом, чтобы *переопределить оператор*, надо объявить специальную функцию — `operator+` (существуют аналогичные названия для большинства остальных операторов).

Теперь реализуем возможность ввода и вывода нашей структуры. Для этого переопределим операторы `«` и `»`:

```cpp
istream& operator>>(istream& stream, Point& p) {
    stream >> p.x >> p.y;
    return stream;
}

ostream& operator<<(ostream& stream, Point& p) {
    stream << p.x << ' ' << p.y;
    return stream;
}
```

Обратите внимание, что надо возвращать сам поток, чтобы можно было считывать несколько значений одной инструкцией.

8 Дополнительные темы

8.1 Что происходит, когда мы запускаем код?

У C++ есть важное отличие от, например, Python или Java. C++ — *компилируемый* язык программирования, в отличие от Python, *интерпретируемого* языка программирования. Программа на C++ сначала переводится в *байт-код* (например, `.exe`-файл), а затем байт-код выполняется операционной системой. Процесс перевода называется *компиляцией*, а программа, его осуществляющая — *компилятором*. Байт-код — это некоторая последовательность примитивных инструкций, которая отличается от *машинного кода* тем, что она зависит только от операционной системы, но не от процессора. В этой универсальности и состоит преимущество байт-кода над машинным кодом.

Программа на Python выполняется сразу из исходного кода, что значительно замедляет этот процесс. Поэтому Python — *интерпретируемый* язык.

С Java все ещё более интересно. Она компилируется в свой байт-код, который не зависит от ОС, а затем этот байт-код выполняется в виртуальной машине (JVM).

8.2 Компиляция через терминал (Linux)

Чтобы установить компилятор `GNU GCC`, введите в терминале `sudo apt-get install g++` (в случае, если в вашей ОС другой пакетный менеджер, используйте его). Чтобы скомпилировать программу, надо ввести команду `g++ <имя файла>.cpp`, где вместо `<имя файла>` надо указать название вашего файла. В таком случае будет создан файл `a.out`. Чтобы выбрать имя для исполняемого файла, укажите флаг `-o <имя исполняемого файла>`, например, `g++ A.cpp -o A`. Существуют и другие флаги. Наиболее распространённые их комбинации приведены в таблице.

`-Wall -Wextra -Wunused -Wconversion`	Включает предупреждения (варнинги). Иногда помогают найти ошибки, так как показывают, например, неиспользованные переменные, неявные преобразования между типами, повторное использование одного и того же названия несколько раз.
`-O3` (латинская O и цифра 3)	Заставляет компилятор как можно сильнее оптимизировать код
`-fsanitize=undefined,bounds,address -g -g3`	Включает *санитайзеры*, которые показывают ошибки во время исполнения (выход за границы массива, переполнение стека и т.д.)
`-DLOC`	Аналогично `#define LOC` в самом начале вашего кода, помогает в дебаге, подробнее см. п. 8.7.2.

8.3 Установка и использование VS Code (Linux)

1. Установите компилятор `GNU GCC`. Для этого введите команду `sudo apt-get install g++` в терминале (в случае, если в вашей ОС нет `apt-get`, используйте менеджер пакетов вашей ОС).

2. Установите VS Code с официального сайта (в случае Ubuntu вы можете установить его из Software Center): https://code.visualstudio.com/.

3. Установите расширение `C/C++ Extension Pack`. Для этого слева нажмите на иконку  , введите название в строке поиска и установите расширение.

4. Запуск программы осуществляется через нажатие иконки  (или клавиши `F5`) или через командную строку (см. п. 8.2).

8.4 Файловый ввод-вывод

В некоторых задачах требуется читать входные данные из одного файла, а записывать в другой. Насколько мне известно, это было популярно в середине 2010-х годов. Познакомимся с двумя способами работать с файлами.

8.4.1 `freopen`

Если вам нужно сделать так, чтобы потоки `cin` и `cout` работали с файлами (*перенаправить потоки ввода-вывода*), то в начале функции `main` нужно дописать:

```
freopen("input.txt", "r", stdin);
freopen("output.txt", "w", stdout);
```

Вместо `input.txt` и `output.txt` нужно использовать имена файлов, указанные в условии задачи.

8.4.2 Потоки `ifstream`, `ofstream`

Этот способ более универсален, так как позволяет работать с большим количеством файлов. Для работы с файлами этим способом требуется добавить строку `#include <fstream>`. Чтобы работать с файлом для чтения и записи соответственно, напишите:

```
ifstream name_in("file_name");
ofstream name_out("file_name");
```

Вместо `file_name` надо указать имя файла, вместо `name_in`, `name_out` — любые имена для этих потоков. Работа с ними осуществляется так же, как и с `cin` и `cout` — через `>>` и `<<`. Приведённые ниже два кода решают задачу нахождения суммы двух чисел, читая ввод из `in.txt` и записывая вывод в `out.txt`:

```
#include <iostream>

using namespace std;

int main() {
    freopen("in.txt", "r", stdin);
    freopen("out.txt", "w", stdout);
    int a, b;
    cin >> a >> b;
    cout << a + b;
}
```

```
#include <fstream>

using namespace std;

ifstream fin("in.txt");
ofstream fout("out.txt");

int main() {
    int a, b;
    fin >> a >> b;
    fout << a + b;
}
```

8.5 Интерактивные задачи

До этого мы сталкивались только со стандартными задачами: весь ввод дан нам в самом начале выполнения программы. Встречается и другой формат задач: *интерактивные задачи*. Их отличие состоит в том, что программа-решение взаимодействует с программой жюри (называемой *интерактором*) через стандартные потоки ввода-вывода. Рассмотрим пример интерактивной задачи:

Задача. Жюри загадало натуральное число от 1 до 10. Угадайте его. Вы можете называть число, а интерактор будет отвечать вам 1, если вы угадали, и 0 иначе. Если вы угадали, Ваша программа должна немедленно завершиться. Вы можете сделать 10 запросов.

```
#include<iostream>

using namespace std;

int main() {
    for (int i = 1; i <= 10; i++) {
        cout << i << '\n';
        cout.flush();
        int x;
        cin >> x;
        if (x == 1) {
            return 0;
        }
    }
}
```

Обратите внимание на строку `cout.flush();`. Она осуществляет *сброс буфера потока вывода*.

Для ускорения вывода C++ выводит данные на экран не сразу при выполнении команды `cout`, а накапливает их в буфере, а затем выводит. Команда `cout.flush()` осуществляет вывод всего содержимого буфера.

Замечание. Интерактивные задачи часто используют идею *бинарного поиска*, с которой вы познакомитесь в следующем году.

8.6 Поиск ошибок с помощью `assert`

При написании большого кода, к сожалению, часто допускаются ошибки. С определением места, где они возникают, может помочь функция `assert`. Чтобы её использовать, нужно подключить библиотеку `cassert`. Функция `assert` принимает аргумент типа `bool`. Если аргумент равен Истине, то продолжается выполнение кода. Если же он равен Лжи, то программа завершается, выдав ошибку. Таким образом, посылка в тестирующей системе получит вердикт Ошибка исполнения, а не Неправильный ответ.

8.7 Как сократить код?

Название типов часто довольно длинные, например, `long long`, `long double`, `vector<int>`. Поэтому часто используются сокращения, например, `ll`, `ld`, `vi`. Чтобы объяснить компилятору, что мы имели в виду, есть несколько способов:

8.7.1 `using`

Пример:

```
#include <iostream>

using namespace std;
using ll = long long;

int main() {
    ll a, b;
    cin >> a >> b;
    cout << a + b;
}
```

Этот подход считается лучшим, но имеет важные ограничения: он позволяет сокращать только типы.

8.7.2 `#define`

Инструкции, начинающиеся с `#`, выполняются во время *препроцессинга*, этапа сборки программы, подготавливающего код к компиляции. Например, во время препроцессинга код всех подключенных библиотек вставляется в исходный код компилируемой программы. Такие инструкции называются *директивами препроцессора*.

Кроме того, во время препроцессинга происходит подстановка `#define`. Если в коде мы укажем директиву

`#define E1 E2`,

то в коде все вхождения `E1` заменятся на `E2`. Выражение `E1` должно быть токеном (т.е. должно быть корректным именем переменной: не содержать пробелов и т.д.), но выражение `E2` может быть произвольным.

Этот способ позволяет сокращать не только названия типов, но и любые названия, например, методов.

```
#include <iostream>
#include <vector>

using namespace std;

#define bg begin

int main() {
    vector<int> v = {5, 4};
    cout << *v.bg(); // 5
}
```

В таком случае говорят, что объявлен *макрос* `bg`.

Бывают и функции-макросы:

```
#include <iostream>

using namespace std;

#define sq(x) ((x) * (x))

int main() {
    cout << sq(5 + 3); // 64
}
```

Запись выше означает, что `sq(x)` будет преобразовано в `((x) * (x))`.

Важно! Скобки лишними не бывают!

Посмотрим, что будет, если убрать скобки:

```
#include <iostream>

using namespace std;

#define sq(x) x * x

int main() {
    cout << sq(5 + 3); // 23
}
```

Вывод изменился. Чтобы разобраться с причинами, посмотрим, во что превратится 8 строчка после замены:

```
    cout << 5 + 3 * 5 + 3;
```

Таким образом, `#define` работает «в лоб», поэтому скобки очень важны.

`#define` часто используется для поиска ошибок в программе (дебага):

```
#include <iostream>

using namespace std;

#define debug(x) cout << #x << " = " << x << '\n';

int main() {
    int x = 5;
    debug(x); // x = 5
    debug(x + 1) // x + 1 = 6
    debug("Hello") // "Hello" = Hello
}
```

В случае объявления макроса, # перед названием его аргумента означает «вставь сюда строку, которая соответствует тому, что было передано в макрос».

В процессе компиляции можно изменять вставляемые блоки кода в зависимости от того, определены ли некоторые константы. Для этого используются директивы `#ifdef`, `#else`, `#endif`.