Содержание

1 Введение в язык C++

1.1 История Языка C++

- C (1969)

  Прародитель языка C++, был создан для упрощения реализации низкоуровневых программ

- C with classes (1979)

  Бьерн Страуструп (Bjarne Stroustrup) решил создать свой языкпрограммирования, который бы сочетал в себе скорость низкоуровневыхязыков и удобство разработки на высокоуровневых.

- C++ (1982)

- C++98 (1998)

  Первый официальный стандарт языка C++

- C++03, C++11, C++14, C++17, C++20, ...

1.2 Современный язык C++

- Неизменно в топе самых популярных и востребованных языков
- C++ используется в ситуациях, когда важна скорость. В частности, приразработке операционных систем, баз данных, серверных и облачныхприложений, утилитах для работы с графикой.
- Обширный набор низкоуровневых инструментов позволяет эффективноиспользовать C++ во встраиваемых системах.
- Что написано на C++? — *Windows, OS X, Google Chrome, Mozila Firefox, YouTube, Photoshop, Telegram*, другие языки программирования ...

1.3 Состав языка C++

1.3.1 Алфавит

Любая программа на языке C++ состоит из следующих символов:

- Цифры (`0` , `1` , ... , `9`)
- Буквы латинского алфавита (`a` , `b` , ... , `z` , `A` , ... , `Z`)
- Нижнее подчеркивание `_`
- Пробельные символы и перенос строки
- Скобки (`{}` , `()` , `[]` , `<>`)
- Кавычки и апострофы (`"` , `'`)

1.3.2 Ключевые слова

Символы могут составлять слова, часть из которых имеют особое значение.

Слова с особым значением названются *ключевыми словами*.

`alignas` , `alignof` , `and` , `and_eq` , `asm` , `auto` , `bitand` , `bitor` , `bool` , `break` , `case` , `catch` , `char` , `char8_t` , `char16_t` , `char32_t` , `class` , `compl` , `concept` , `const` , `consteval` , `constexpr` , `constinit` , `const_cast` , `continue` , `co_await` , `co_return` , `co_yield` , `decltype` , `default` , `delete` , `do` , ...

Полный список ключвых слов можно найти по ссылке

1.3.3 Идентификаторы

- *Идентификатор* — имя программного объекта.
- Идентификатор представляет собой последовательность букв, цифр, нижнихподчеркиваний, составленную по следующим правилам:

1. Не начинается с цифры
2. Не содержит пробелов
3. Не является ключевым словом
4. Чувствительно к регистру (`Alice` и `alice` — разные идентификаторы)

Примеры:

```
alla                //OK
allo4ka             //OK
allochka-ivanova    //CE (character '-' cannot be in identifier)
allo4ka_ivanova     //OK
allochka ivanova    //CE (must not contain whitespace characters)
2dima               //CE (must not start with a number)
_                   //OK
_2dima              //OK
asm                 //CE (key word)
AsM                 //OK
```

1.3.4 Константы/литералы

Константы/литералы — значения встроенные в текст программы.

- Целочисленные константы:
  - Десятичные `123`
  - Восьмеричные `045`
  - Шестнадцатеричные `0x1A` , `0XBD`
  - Двоичные (C++14) `0b101` , `0B1001`
- Символьные константы — `'a'` , `'n'`

- Вещественные константы — `3.14` , `.24e10`
- Строковые литералы: `"a"` , `"c"` , `"n"` , `"This␣is␣string"`
- Логические константы — `true` , `false`
- Нулевой указатель — `nullptr`

1.3.5 Знаки операций

Каждая операция(operator) характеризуется *арностью, приоритетом и ассоциативностью.*

- Арность — количество аргументов, которое может принимать операций.
- Приоритет — свойство, которое определяет порядок вычисления операций.
- Ассоциативность — свойство, которое определяет порядок выполнения операций с одинаковым приоритетом

Важно: Один и тот же знак операции может иметь разный смысл, арность и приоритет в зависимости от количества операндов, типа данных.

Например:

```
10 / 3;    //Integer division - 3
-10 / 3.; //Real division - 3.3333
-5;        //Unary operation
5 - 3;     //Binary operation
```

Более подробно про операции можно прочитать по ссылке

1.4 Запуск программ на C++

- Язык C++ — *компилируемый* язык. Это означает, что код (текст программы) сначала обрабатывается специальной программой (компилятор), которая переводит его в машинные инструкции и создает исполняемый файл. Далее этот файл может быть запущен и исполнен как любая другая программа.

  Примерами таких языков являются — *C, C++, Basic, Go, Pascal, Rust, ...*

- Также существуют *интерпретируемые* языки программирования. В них текст программы подается на вход другой программе (интерпретатор), который сразу начинает исполнение написанного кода без этапа сборки исполняемого файла.

  Примерами таких языков являются — *Python, PHP, Perl, Ruby*

- В некоторых языках (например, *Java*) используется гибридный подход.

Кодом на C++ является любой текстовый файл, написанный по правилам языка C++.

Чтобы сделать из него программу, необходимо его *скомпилировать*, то есть передать код программе-компилятору.

После этого появится программа (файл), которую можно запустить с помощью командной строки:

```
g++ <files name> -o <programms name>
```

1.5 Первая программа

Минимальная программа, которая ничего не делает (почти), на C++ выглядит так:

```
int main() {}
```

Данная программа состоит из ключевого слова (`int`), идентификатора (`main`), скобок (`()` , `{}`)

Пояснение: здесь происходит определение функции с именем `main` , которая ничего не принимает(`()`), возвращает целое число(`int`) и не делает ничего (`{}`). Функция с таким именем обязана присутствовать в любой программе, так как именно с нее начинается исполнение программы.

Функция `main` по умолчанию возвращает `0` . Но это значение можно указать явно:

```
int main() {
  return 0; // Key word return + constant (literal) 0
}
```

Кроме того, мы можем выполнять целочисленную арифметику в C++ и возвращать ее результат.

```
int main() {
  return 2 + 2 ; 2;
}
```

Примечание: сначала вычисляется результат выражения справа от `return` , а затем это значение возвращается из функции `main` .

1.6 Структура программы

1.6.1 Объявления

Любая программа на C++ — это последовательность *объявлений.*

Объявление (declaration) — введение (создание) некоторой именованной сущности.

Пример:

```
int main() { // this is a function declaration "main"
  return 0;
}
```

Объявление функции `main` можно сократить:

```
int main();
```

Такое объявление является объявлением без определения.

1.6.2 Определение

В общем случае объявления лишь вводят новые имена, но не дают им конкретного смысла.

Определение (definition) — объявление с полной информацией о создаваемой сущности.

Пример:

```
int main(); // This is the function declaration
int main() { // this is the function definition
  return 0;
}
```

Важно: в программе должно быть ровно одно определение функции `main` . Нельзя не определять `main` .

1.6.3 Директива

Помимо объявлений в программе могут встречаться директивы препроцессора.

Директивы препроцессора — вспомогательные инструкции, которые начинаются с `#` и осуществляют манипуляции над текстом программы и опциями сборки

Примеры:

```
/* Paste the text file "file" from the current folder to the
given location in the code */
#include "file"
/* Paste the text file "file" from a specific folder at the
given location in the code*/
#include <file>
// Replace all occurrences of text "X" with text "Y"
#define X Y
```

1.6.4 Операторы (инструкции)

Функции (в частности, функция `main`) описывают последовательность действий, которые выполняются по порядку, сверху-вниз.

Оператор (инструкция, statement) — фрагмент кода, описывающий некоторое действие.

На данный момент мы знакомы с одним оператором — оператором возврата из функции `return` .

1.6.5 Выражения

Некоторые действия (операторы) требуют проведения вычислений. За вычисления в языке C++ отвечают *выражения.*

Выражение (expression) — последовательность операций и их операндов, задающая вычисление.

У любого выражения есть результат (может быть пустой), который "подставляется"вместо него.

```
int main() {
  return 2 + 2 * 2; // 2 + 2 * 2 - expression with result 6.
}
```

1.6.6 Комментарии

Иногда в коде хочется сделать некоторое примечание или напоминание о том, что означает тот или иной блок кода. Но мы знаем, что просто так написать текст в коде нельзя, так как,

скорее всего, он не будет удовлетворять правилам грамматики языка C++. Для этих целей в языке есть *комментарии*.

Комментарий — текст кода, который игнорируется при сборке программы.

В C++ есть два типа комментариев:

```
// single line comment
/* multiline
comment */
```

1.7 Вывод результата в консоль

Все программы, которые были написаны нами ранее, возвращали результат операционной системе в виде "кода ошибки":

```
int main() {
  return 2 + 2 * 2;
}
```

Это совсем неудобно, так как

1. Результатом работы программы не всегда является целое число
2. Коды ошибок могут принимать лишь значения от 0 до 255
3. Результат приходится получать отдельно с помощью специальной переменной окружения — `echo $?`

Для вывода результата на экран в C++ обычно используется *потоковый вывод*:

```
#include <iostream> // 1
int main() {
  std::cout << 2 + 2 * 2; // 4
  return 0; // 0 - no errors
}
```

- строка 1:

  Знаем, что `#include` — директива, означающая "вставку"другого файла в текст нашей программы. В файле `iostream` (Input-Output STREAM) сожержится все необходимое для работы с выводом.

- строка 4: В этой строке записано **выражение** вывода результата `2 + 2 * 2` в поток `std::cout` (Console OUTput).

В поток вывода можно записывать и другую информацию, например, строки. Хорошим тоном является вывод переноса строки в конце результата (для более красивого сообщения на экране). За это отвечает специальный символ `'n'` .

Пример:

```cpp
#include <iostream>
int main() {
  std::cout << "2␣+␣2␣*␣2␣=␣";
  std::cout << 2 + 2 * 2;
  std::cout << '\n';
  return 0;
}
```

Весь вывод можно объединить в одно выражение:

```cpp
#include <iostream>
int main() {
  std::cout << "2␣+␣2␣*␣2␣=␣" << 2 + 2 * 2 << '\n';
  return 0;
}
```

1.8 Самая важная программа

Используя накопленные знания, напишем программу, приветствующую мир:

```cpp
#include <iostream>
int main() {
  std::cout << "Hello,␣world!\n";
  return 0;
}
```

Примечание: обратите внимание, что символ переноса строки может стоять внутри строки, не обязательно выводить его отдельно

2 Фундаментальные типы данных

2.1 Типы данных

Тип данных — свойство программных сущностей, которое определяет:

1. Множество допустимых значений
2. Допустимые операции
3. Низкоуровневое представление (интерпретация битов)

В C++ существует большое количество типов данных. Более того, можно (и мы будем) создавать и свои типы. Однако каждый тип данных состоит из, или является производным от одного из фундаментальных типов:

- "Пустой"тип — `void`
- Целые тип — `int` , `short` , `long` , `long long`
- Числа с плавающей точкой — `float` , `double` , `long double`

- Символьный тип — `char` , `char8_t` , `char16_t` , `char32_t`
- Логический тип — `bool`
- Нулевой указатель — `std::nullptr_t`

2.2 Целый тип данных

Самым используемым типом данных является целочисленный тип. Поэтому ему мы уделим больше всего внимания.

2.2.1 int

Основным целочисленным типом является `int` .

Стандарт C++ гарантирует, что размер `int` **как минимум** 16 бит (2 байта).

Это значит, что значения типа `int` могут лежать в диапазоне $[-32'768; 32'767]$.

На практике, как правило, `int` занимает 32 бита (4 байта), то есть принимает значения в диапазоне $2'147'483'648; 2'147'483'647$

.

Литералами типа `int` являются — 0, 1, -4, 100, $576'843$

2.2.2 long

Стандарт C++ гарантирует, что размер `long` **как минимум** 32 бита (4 байта) и **не меньше** размера `int` .

Это значит, что значения типа `long` могут лежать в диапазоне $[2'147'483'648; 2'147'483'647]$.

Литералами типа `long` являются — $0l$, $1L$, $-4l$, $100L$, $1'123'576'843l$

2.2.3 long long

Стандарт C++ гарантирует, что размер `long long` **как минимум** 64 бита (8 байт) и **не меньше** размера `long` .

Это значит, что значения типа `long long` могут лежать в диапазоне $[9'223'372'036'854'775'808; 9'223'372'036'854'775'807]$.

Литералами типа `long long` являются — $0ll$, $1LL$, $-4ll$, $100LL$, $1'123'576'843ll$

2.2.4 short

Стандарт C++ гарантирует, что размер `short` **как минимум** 16 бит (2 байта) и **не больше** размера `int` .

Это значит, что значения типа `short` могут лежать в диапазоне $[-32'768; 32'767]$.

Собственных литералов тип `short` не имеет.

2.2.5 Модификаторы signed/unsigned

По умолчанию все целые типы *знаковые*, то есть могут хранить как положительные, так и отрицательные числа. Чтобы это подчеркнуть, к названиям типов можно приписывать модификатор `signed` :

```
int == signed int == signed
long == signed long == long int == signed long int
long long == signed long long == long long int == signed long long int
short == signed short == short int == signed short int
```

Если предполагается, что значение должно хранить только неотрицательные числа, то можно к имени типа добавить `unsigned` .

В этом случае допустимые значения меняются следующим образом:

- 2 байта — $[0; 65'535]$
- 4 байта — $[0; 4'294'967'295]$
- 8 байт — $[0; 18'446'744'073'709'551'615]$

```
unsigned int == unsigned
unsigned long == unsigned long int
unsigned long long == unsigned long long int
unsigned short == unsigned short int
```

Беззнаковые литералы — $1u$, $4ul$, $6ull$

2.2.6 Операции

Над целыми числами (кроме short !) можно выполнять все арифметические операции. Правила выполнения операций звучат следующим образом:

- Результатом операции является значение того же типа, что и у операндов
- При переполнении беззнаковых чисел выполняется арифметика по модулю размера этого типа
- В остальных ситуациях переполнение — это *Undefined Behavour*(неопределенное поведение)
- Деление на 0 — *Undefined Behavour*
- Битовые сдвиги отрицательных чисел тоже могут приводить к UB

Если же операнды имеют разные типы (или тип `short`), то:

- (`signed` / `unsigned`) `short` приводится к `int`
- Менее широкий тип приводится к более широкому
- Знаковый тип приводится к беззнаковому

Примеры:

```
0 + 0l; // long
0ll + 0; // long long
0l + 0ll; // long long
0u + 0; // unsigned int
0 + 0ul; // unsigned long
0ul + 0ll; // unsigned or signed long long
```

2.2.7 Целые типы фиксированной ширины

На разных системах целые типы могут иметь разные размеры, что осложняет жизнь.

Для решения этой проблемы в C++11 появились типы *фиксированной ширины*:

```
int8_t, int16_t, int32_t, int64_t
uint8_t, uint16_t, uint32_t, uint64_t
```

Они имеют **в точности** тот размер, который указан в названии. Чтобы использовать их, необходимо подключить заголовочный файл `<cstdint>`

2.3 Символьные типы

Символьные типы данных используются со строками. Они будут рассмотрены далее в курсе, но пока необходимо с ними познакомиться.

2.3.1 char

Тип `char` используется для хранения символов.

Символ представляется некоторым 1 байтовым целым числом согласно ASCII таблице

Символьные литералы — `'a'`, `'1'`, `'n'`, `'@'`, ...

Так как `char` — целое число, оно может быть как знаковым, так и беззнаковым (какой из типов используется по умолчанию - зависит от реализации).

2.3.2 char_t, char16_t, char32_t

Широкие символьные типы используются для хранения символов из кодировок $UTF-16$ и $UTF-32$. Более подробно по ссылке.

2.3.3 Арифметика

Так как символы представляются целыми числами, тип char можно считать 8-битным целым числом и использовать все арифметические операции.

`'E' + ('a' - 'A') == 'e'`

Напоминание: при выполнении арифметики числа, у которых тип имеет ранг меньший int , приводятся к int .

Примеры:

```
'a' + 0; // int
'a' + 0l; // long
'a' + 0ll; // long long
'a' + 'a'; // int
```

2.4 Логический тип

2.4.1 bool

Объекты логического типа могут принимать всего 2 значения: `true` / `false` .

Имеет размер в 1 байт.

Обычно используется для хранения результата сравнения:

```
5 > 6;  // false
5 < 6;  // true
5 >= 6; // false
5 <= 6; // true
5 == 6; // false
5 != 6; // true
```

Объекты типа `bool` могут выступать операндами логических операций:

```
5 > 6 && 5 < 6; // false (logical "and")
5 > 6 || 5 < 6; // true (logical "or")
!(5 > 6); // true (logical "not")
```

Также в C++ возможно использование специальных слов `and` , `or` , `not` . Но это является плохим стилем, так как эти ключевые слова пришли с языка C, где первоначально не было специальных символов. Использовать их не рекомендуется, а на нашем курсе — запрещено.

`&&` и `||` — особенные операции:

1. Гарантируется, что выражение слева будет вычислено до выражения справа.
2. Если слева значение `true` , то правая часть `||` вычисляться не будет.
3. Если слева значение `false` , то правая часть `&&` вычисляться не будет.

```
5 < 6 || ... ; // whatever is on the right is not evaluated (at all)
5 > 6 && ...; // whatever is on the right is not evaluated (never)
```

2.4.2 Арифметика

Логический тип в C++ является разновидностью целового типа и может быть использован в арифметических выражениях. При этом `true == 1` , а `false == 0` :

```
true + 5; // int: 6
10l * false; // long: 0
```

Верно и обратное: при подстановке в логическую операцию ненулевое значение интерпретируется как `true`, нулевое — `false`:

```
5 && 1; // true
0 || 0; // false
!-1;    // false
```

2.5 Числа с плавающей точкой

Числа с плавающей точкой используются для хранения рациональных чисел.

`float` — числа с одинарной точностью (4 байта, примерно $[\pm 10^{-38}; \pm 10^{38}]$)

`double` — числа с двойной точностью (8 байт, примерно $[\pm 10^{-308}; \pm 10^{308}]$)

`long double` — числа с расширенной точностью (12 или 16 байт, примерно $[\pm 10^{-4932}; \pm 10^{4932}]$)

Вещественные литералы:

```
0., 1.5, 3.14159; // double
0f, 1.5F, 3.14159f; // float
0l, 1.5L, 3.14159l; // long double
123.456e10; // 123.456 * 10^10
123.456e-10; // 123.456 * 10^(-10)
```

2.5.1 Особенности

- Можно применять те же арифметические операции, что и к целым числам (кроме битовых операций и взятия остатка)

  `0.1 + 0.2 * 5.67 / 0.9; //real division`

- Следует помнить, что дробные числа имеют ограниченную точность при вычислениях:
  `0.1 + 0.2 != 0.3;`

- Числа с плавающей точкой всегда знаковые.

- Имеются специальные значения: `+inf`, `-inf`, `nan`:

  ```
  1. / 0;  //inf
  -1. / 0; //-inf
  0. / 0;  //nan
  ```

- Если типы чисел с плавающей точкой не совпадают, то менее широкий аргумент приводится к более широкому:

  ```
  5.0 + 1.5f; // double
  5.0 + 1.5l; // long double
  ```

- При выполнении арифметической операции над целым и дробным числом целое число приводится к типу дробного:

  `1 + .0f; //float`

2.6 Пустой тип: void

Тип `void` — тип с пустым множеством значений.

Главное применение — сообщить о том, что выражение ничего не возвращает (результата нет).

3 Переменные

3.1 Модель памяти

Память компьютера имеет довольно сложное техническое устройство. Кроме того, методы управления памятью могут сильно отличаться в зависимости от используемой архитектуры и ОС.

Модели памяти предоставляют удобную абстракцию для работы, которая в большинстве ситуаций позволяет писать код, не задумываясь о низкоуровневых деталях реализации взаимодействий.

При написании кода мы руководствуемся предоставленной моделью памяти, а реализация и исполнение конкретных инструкций лежит на плечах компилятора/ интерпретатора.

3.2 Модель памяти C++

- Модель памяти C++ представляет из себя *последовательность нумерованных ячеек памяти*, которые называются *байтами*.
- Номер ячейки памяти называется *адресом*.
- Таким образом, *байт* — минимальная адресуемая единица памяти.
- Стандарт C++ гарантирует, что размер байта в точности совпадает с размером объектов типа *char*.
- Все ячейки памяти равноправны. За исключением нулевой ячейки: туда ничего нельзя записать и оттуда ничего нельзя прочитать.
- Последовательности ячеек могут иметь различную интерпретацию в зависимости от того, элемент какого типа мы рассматриваем в данный момент.

3.3 Переменные

Переменная — именованная область памяти. Чтобы создать переменную, необходимо указать ее тип, дать ей имя и, возможно, начальное значение:

```
int x;
short y = 1; // 1 is cast to short
float z(1.5); // 1.5 is cast to float
char t{'@'};
```

Определение переменной выделяет область памяти достаточную для хранения объектов указанного типа и закрепляет за этой областью обозначенное имя.

Однотипные переменные можно создавать в одну строку *[плохой стиль]*.

```
int x = 1, y, z = 3;
```

Переменные можно использовать в выражениях, их значения будут использованы для вычислений:

```
int main() {
  int x = 1;
  int y = 2;
  std::cout << x + 3 * y << '\n'; return 0;
}
```

Как и любую другую сущность, переменную нельзя использовать до того, как она была объявлена:

```
int main() {
  std::cout << x + 3 * y << '\n';
  int x = 1;
  int y = 2;
  return 0;
}
```

```
main.cpp: In function   int   main()   :
main.cpp:4:16: error:    x    was not declared in this scope
  4 | std::cout << x + 3 * y << '\n';
    |              ^
main.cpp:4:24: error:    y    was not declared in this scope
  4 | std::cout << x + 3 * y << '\n';
    |                      ^
```

3.3.1 Автоматическая область действия

Блоки бывают полезны для структурирования кода функции на логические части, а также при использовании с другими операторами (об этом позже в курсе).

Важное свойство блоков состоит в том, что они завершают действие переменных, объявленных внутри блока.

```
int main() {
  int x = 1;
  {
    int y = 2;
    std::cout << x + 3 * y << '\n';
  } // <-- at this point 'y' no longer exists
  return y; // error: there is no 'y' in this block
}
```

Таким образом, получаем, что обычная переменная, объявленная внутри функции, действует с места объявления до *ближайшего* конца блока, внутри которого она объявлена.

3.3.2 Глобальная область действия

Переменные можно объявлять и вне функций. Такие переменные называются *глобальными*.

```cpp
int x = 1;
int main() {
  int y = 2;
  std::cout << x + 3 * y << '\n';
  return 0;
}
```

Глобальные переменные существуют на протяжении всего времени работы программы, но могут быть использованы только после объявления!

Чтобы использовать переменные, объявленные после функции, необходимо объявить переменную `x` внутри функции и указать, что эта переменная определена в другом месте. За это отвечает ключевое слово `extern` :

```cpp
int main() {
  int y = 2;
  extern int x; // declaration without definition
  std::cout << x + 3 * y << '\n';
  return 0;
}
int x = 1; // definition x
```

3.3.3 Классы памяти

В C++ каждый объект в памяти принадлежит одному из 4х классов:

- Автоматическая (стековая) память. Память, в которой хранятся обычные локальные переменные функции. Живут до конца своего блока.
- Глобальная (статическая) память. Память, которая выделяется в начале работы программы (до старта main) и освобождается при завершении.
- Динамическая память (куча). Память выделяется и освобожается вручную. Обсудим позже.
- Потоковая память. Связана с потоком исполнения. Выделяется в начале выполнения потока, освобождается при завершении. В курсе не обсуждается.

3.3.4 Область видимости (scope)

Область видимости (scope) переменной — часть кода, из которой возможен обычный доступ к переменной по ее имени.

Очевидно, что область видимости не превышает области действия переменной.

При этом *scope* может быть меньше области действия:

```cpp
int x = 0;
int main() {
  int x = 1;
  std::cout << x << '\n'; // 1
  return 0;
}
```

В этом примере, область действия обоих `x` распространяется на функцию `main`. Однако внешнее имя скрыто локальным именем.

Для того, чтобы получить доступ к глобальной переменной, можно воспользоваться операцией разрешения области видимости (`::`)

```
int x = 0;
int main() {
  int x = 1;
  std::cout << ::x << '\n'; // 1
  return 0;
}
```

`::` перед именем переменной говорит компилятору о том, что имя нужно искать в глобальной области.

Однако в случае:

```
int x = 0;
int main() {
  int x = 1;
  {
    int x = 2;
    std::cout << x << '\n'; // 2
    std::cout << ::x << '\n'; // 0
  }
  return 0;
}
```

Способа обратиться к промежуточному `x` по его имени нет.

3.3.5 Инициализация переменных

Переменные при определении могут быть инициализированы.

```
int a; // uninitialized variable
int b = 1;
int c(2);
int d{3};
int e{}; // empty curly braces = padding with zeros
```

Последние 4 строки так или иначе инициализируют значение переменной (делают запись в соответствующую область памяти).

Чтобы изменить значение переменной, можно воспользоваться операцией присваивания:

```
int x;
x = 10; // changed the value to 10
int y = 1;
y = 2; // changed the value to 2
```

Важно: инициализация с помощью `=` **НЕ** является присваиванием.

Чтение из неинициализированной переменной приводит к UB. Пользоваться ей можно только после установки конкретного значения.

3.3.6 Потоковое чтение данных

Для чтения значений из потока можно воспользоваться библиотекой `<iostream>` :

```
#include <iostream>
int main() {
  int x;
  int y;
  std::cin >> x >> y;
  std::cout << x + y << '\n';
  return 0;
}
```

Данная программа получает с консоли 2 целых числа и выводит их сумму.

3.3.7 Квалификатор const

Константный объект — объект, к которому применимы только те операции, которые не меняют его логического состояния.

Чтобы объявить константную переменную, необходимо к названию типа добавить слово `const` :

```
const int x = 0; // More preferred option
int const y = 0;
```

Теперь попытка изменения `x` будет приводить к ошибке компиляции:

```
x = 1; // CE
std::cin >> y; // CE
```

3.3.8 Квалификатор volatile

К типу переменной можно дописать ключевое слово `volatile` . Это означает, что операции с этой переменной не должны быть оптимизированы (удалены, переставлены местами с другой операцией и т.п.).

Это бывает полезно, когда доступ к переменной осуществляется не на прямую, а опосредовано.

```
volatile int x = 0;
int volatile y = 0;
```

3.4 Больше о выражениях

3.4.1 Присваивающие операции

Для изменения значения переменной используется присваивание (`=`):

```
y = 5;
x = y;
x = y * y;
```

Помимо обычного присваивания, существуют присваивающие арифметические операции:

```
x += 5; // x = x + 5
y %= 2; // y = y % 2
y |= 9; // z = z | 9
```

Присваивающие операции не только изменяют значение, но и, как и другие, возвращают результат своего исполнения — новое значение.

```
std::cout << (y = 5);      // 5
std::cout << (x = y);      // 5
std::cout << (x = y * y); // 25
```

```
std::cout << (x += 5); // 30
std::cout << (y %= 2); // 1
std::cout << (y |= 9); // 9
```

Это дает возможность строить цепочки присваиваний:

```
x = y = 5;
z = x = y *= y;
```

Более того, оказывается, что результатом присваивающих операций является не просто значение, а как бы сама исходная переменная.

То есть, в результат можно снова что-то присвоить!

```
(x = y) = 5;  // x == 5
(x = y) *= 0; // x == 0
```

При этом заметим, что в качестве левого операнда присваивания не может стоять литерал или результат неприсваивающей операции:

```
5 = x;
x * y = 10;
```

Сборка падает со странной ошибкой:

```
main.cpp:8:3: error: lvalue required as left operand of assignment
    8 | 5 = x;
      | ^
main.cpp:9:5: error: lvalue required as left operand of assignment
    9 | x * y = 10;
      | ~~^~~
```

3.4.2 Резлультат выражения

Результат работы выражения проявляется в

1. Возвращаемом значении
2. Побочных эффектах

Каждое выражение характеризуется типом **возвращаемого значения** и **категорией значения**.

- С типом возвращаемого значения все ясно (если нет, то вернитесь к разделу с фундаментальными типами)
- Категория значения отвечает на вопрос: материален ли результат выражения, то есть, существует ли он в виде объекта в памяти.

```
int x, y =0; // type | side effect | value category
// ---------------------------------------------------
x + y;       // int   | no           | not material
x = 5;       // int   | x stores 5   | material (x)
x *= 2;      // int   | x stores 10  | material (x)
y + (x *=6); // int   | x stores 60  | not material
x / 2.;      // double| no           | not material
```

3.4.3 Категории значений

Материальная категория значений называется *lvalue*, нематериальная — *prvalue*.

Так как *lvalue* значения материальны (соответствуют некоторому расположению в памяти), они могут быть использованы в качестве левого операнда в операцииприсваивания (за некоторыми исключениями).

Теперь природа ошибок становится ясной:

```
main.cpp:8:3: error: lvalue required as left operand of assignment
    8 | 5 = x;
      | ^
main.cpp:9:5: error: lvalue required as left operand of assignment
    9 | x * y = 10;
      | ~~^~~
```

Примечание: помимо *lvalue* и *prvalue* встречается *xvalue*, но о нем будет говорить позже.

3.4.4 Инкремент и декремент

Инкремент/декремент — унарные операции, осуществляющие увеличение/уменьшение аргумента на 1.

Как и присваивание, могут быть применены только к *lvalue*.

```
int x = 0;
x++; // postfix increment (x = 1)
++x; // prefix increment (x = 2)
x--; // postfix decrement (x = 1)
--x; // prefix decrement (x = 0)
```

Если же мы попробуем сделать инкеремент/декремент не *lvalue*, например:

```
int x = 0;
++5;
(x + 1)--;
```

Мы получим следующие ошибки:

```
main.cpp:3:5: error: lvalue required as increment operand
    3 | ++5;
main.cpp:4:6: error: lvalue required as decrement operand
    4 | (x + 1)--;
```

Пост и пре инкременты/декременты отличаются друг от друга возвращаемым значением.

Префиксные операции `++` и `--` возвращают обновленное *lvalue* значение (то есть сам аргумент).

Постфиксные версии возвращают старое значение (до обновления) *rvalue*.

```
int x = 0;
std::cout << ++x << '\n'; // 1 (x == 1)
std::cout << x++ << '\n'; // 1 (x == 2)
--x = 10; // x == 10
x-- = 11; // CE
```

3.4.5 Операция(operator) sizeof

`sizeof` — унарная операция(operator), возвращающая размер объекта в байтах.

- Если `sizeof` применяется к *типу*, то возвращает, сколько памяти в байтах занимают элементы этого типа

```
sizeof(char) == 1; // always
sizeof(int) == 4;  // probably
```

- Если `sizeof` применяется к выражению, то вычисляет размер типа возвращаемого значения.

  Важно: `sizeof` не вычисляет переданное выражение, а лишь анализирует.

```
long long x;
sizeof('a'); // 1
sizeof(1 / 0); // 4
sizeof(x = 1); // 8 (x does not change!)
sizeof(++x); // 8 (x does not change!)
```

3.4.6 Операция(operator) static_cast

`static_cast` принудительно осуществляет допустимые неявные преобразования значений одного типа в другой:

`static_cast<new type> (expression)`

```
int x = 2;
int y = 2000000000;
std::cout << x / y << '\n'; // integer division
std::cout << static_cast<float>(x) / y << '\n';

std::cout << x * y << '\n'; // overflow (UB)
std::cout << static_cast<int64_t>(x) * y << '\n';
```

3.4.7 Операция(operator) ,

операция (operator) "запятая"позволяет объединить несколько выражений в одно:

```
++x, y--, std::cout << x + y, 5;
```

- операция(operator) `,` гарантирует, что сначала будет вычислено выражение слева.
- Результатом операции `,` является результат второго операнда (правый операнд).

3.5 Ключевое слово auto

- До C++11 `auto` использовалось для обозначения переменной в автоматической области памяти (локальная переменная).
- Возможность оказалась настолько неактуальной, что в стандарте C++11 кардинально поменяли смысл слова `auto` (довольно исключительная ситуация).
- Прежний смысл отдали ключевому слову `register` .
- Однако в C++17 отказались и от него, и теперь `register` считается устаревшим (слово зарезервировано для дальнейшего использования).

В наше время `auto` используется для автоматического вывода типа переменной при инициализации

```
int main() {
  auto x = 0; // int
  auto y = 0.0; // double
  auto px = &x; // int*

  auto z; // CE: variable type cannot be determined
}
```

Очень удобно, когда речь идет о длинных именах типов.

Правила вывода типа `auto` совпадают с правилами вывода для шаблонных параметров (за некоторым исключением).

```cpp
int x = 0; const int cx = 1; int& rx = x; int arr[10];

auto y = x; // int
auto cy = cx; // int
auto ry = rx; // int
auto arr_y = arr; // int*

auto& z = x; // int&
auto& cz = cx; // const int&
auto& rz = rx; // int&
auto& arr_z = arr; // int(&)[10]

//Exception
auto x = {1, 2, 3}; // std::initializer_list<int>
```

4 Условная операция(operator) и условный оператор(statment)

4.1 Условная операция(operator)

В языке C++ существует единственная тернарная операция(operator) (принимающая 3 аргумента) — условная операция(operator) (`?:`). Имеет вид `<bool-expr> ? <expr1> : <expr2>` , где:

1. `<bool-expr>` — выражение со значением конвертируемым в bool
2. `<expr1>` , `<expr2>` — выражения с "совместимыми"возвращаемыми значениями.

Если `<bool-expr>` возвращает `true` (или то, что приводится к true), то выполняется `<expr1>` , иначе — `<expr2>` .

Решим с помощью условной операции задачу:

На вход поступают неотрицательные целые числа x и y, найти результат целочисленного деления x на y, если $y \neq 0$, а иначе вывести 1.

```cpp
#include <iostream>

int main() {
  int x;
  int y;
  std::cout << (y != 0 ? x / y : -1) << '\n';
  return 0;
}
```

Проблем с UB нет, так как `x/y` выполняется **только** при `y != 0` .

Попробуем аналогично решить другую задачу:

На вход поступают неотрицательные целые числа x и y, найти результат целочисленного деления x на y, если $y \neq 0$, а иначе вывести "Error".

```cpp
#include <iostream>

int main() {
  int x;
  int y;
  std::cout << (y != 0 ? x / y : "Error") << '\n';
  return 0;
}
```

Получим сообщение с ошибкой:

```
main.cpp:6:24: error: operands to '?:'
have different types 'int' and 'const␣char*'
6 | std::cout << (y != 0 ? x / y : "Error") << '\n';
```

Проблема заключается в том, что типы `int` и "строка"несовместимы, то есть не существует неявного преобразования одного в другое.

4.1.1 Возвращаемый тип

```
<bool-expr> ? <expr1> : <expr2>
```

Коротко, результатом условной операции является наибольший тип, способный вместить результат `<expr1>` и `<expr2>` .

Результирующий тип, разумеется, не зависит от истинности `<bool-expr>` , так как разрешение типов происходит на этапе компиляции, а не во время выполнения:

```cpp
x > 0 ? 1 : 1.0; // return type - double
true ? 0 : 5ll; // return type - long long
false ? "str" : 0; // error - "string" and int incompatible
```

4.1.2 Категория значения

Правила выбора категории значения логической операции:

- Если типы и категории значений обоих выражений совпадают, то выбора нет.
- Если категории значения разные, то результат — prvalue
- Если категории совпадают, а типы отличаются только наличием дополнительного cv-квалификатора, то результат — общая категория + тип с дополнительным cv-квалификатором.
- Если типы разные (даже с точностью до cv), то результат — наиболее подходящий общий тип категории prvalue.

4.2 Условный оператор

Условный оператор(statment) в C++ имеет вид:

```
if ([init] <condition>) <statement-true> [else <statement-false>]
```
,
где:

- `condition` — либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool` .
- `statement-true` — оператор, который выполняется, если `condition` — `true`
- `statement-false` — оператор, который выполняется, если `condition` — `false`
- `init` — либо выражение, либо объявление. Область действия объявленной сущности совпадает с областью условного оператора.

4.2.1 Примеры

Примерами простого использования `if` являются:

```
if (x > 0) std::cout << x << '\n';
```

```
if (int x = 0) { // false
  int y;
  std::cin >> y;
  std::cout << y / x << '\n';
} // here x and y are no longer valid
```

```
if (x != 0 && y / x > 5) return y / x - 5;
```

```
if (x);
```

Напоминание: в `condition` новую переменную инициализировать обязательно:

```
if (int x) std::cin >> x; // error: x = ?
```

4.2.2 Примеры (init)

Примерами использования `if` с инициализацией являются:

```
if (const int x = y + z; x != 5) std::cout << x << '\n';
```

```
if (int x; x = 0) { // false (in init initialization is optional)
  int y;
  std::cin >> y;
  std::cout << y / x << '\n';
}// here x and y are no longer valid
```

```
// init can contain any expression (not just a declaration)
if (std::cin >> x; x != 0 && y / x > 5) return y / x - 5;
```

4.2.3 Примеры (else)

Примерами использования `if` с `else` являются:

```
if (x != 0) std::cout << y / x << '\n';
else std::cout << "Error\n";
```

```
if (x >= 0 && y >= 0) std::cout << "Positives\n";
else {
  int z = x * y;
  std::cout << (z > 0 ? "Negatives\n" : "Different\n");
}
```

```
if (x >= y) {
  if (x >= z) {
    std::cout << x << '\n';
  } else {
    std::cout << z << '\n';
  }
} else if (y >= z) std::cout << y << '\n';
else std::cout << z << '\n';
```

`else` относится к ближайшему неспаренному `if` !

4.3 Оператор(statment) switch

4.3.1 Определение

оператор(statment) switch представляет собой оператор(statment) выбора нужной ветви в зависимости от значений выражения или определения. Он выглядит следующим образом:

`switch ([init] <condition>) statement`

- `init` — либо выражение, либо объявление. Область действия объявленнойсущности совпадает с областью оператора.
- `condition` — выражение или определение, имеющее целый илиперечислимый тип.
- `statement` — произвольный оператор.

Как правило, в качестве `statement` выступает составной оператор, в котором присутствуют метки `case` , `default` и операторы `break` :

```
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  default: std::cout << "Other\n";
}
```

Конструкция `case` и `default` выглядит следующим образом:

`case <const-expr>: <statements>`

`default: <statements>`

- `const-expr` — выражение с целым или перечислимым значением, которое вычислимо **на этапе компиляции**.
- `statements` — последовательность операторов.

После вычисления `switch` условия ищется соответствующая метка и управление передается ее первому оператору.

Если нужного значения найдено не было, то осуществляется переход к `default` .

Важно!

После перехода к метке выполняются все операторы, расположенные после нее (даже те, которые лежат под другими метками).

Чтобы завершить выполнение операторов, необходимо написать оператор(statment) `break;`

4.3.2 Примеры без инициализации

Примерами использования `switch` без инициализации являются:

```
int x = ...;
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n";
  //after "One" will display "Default"
  default: std::cout << "Other\n";
}
else std::cout << "Error\n";
```

```
int x = 0;
const y = 2;
switch (x * y) {
  case 2 * 2: ... // OK
  case y: ...     // OK
  case x * 2: ... // CE (not evaluated at compile time)
}
```

Напоминание: в качестве условия может стоять только целое число:

```
double x = ...;
switch (x) { //CE
  ...
}
```

4.3.3 Примеры с инициализации

Примерами использования `switch` с инициализацией являются:

```cpp
switch (int x = ...) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  default: std::cout << "Other\n";
}
```

```cpp
switch (std::cin >> x; x * x) {
  case 0:
  case 1: std::cout << "x␣*␣x␣<=␣1\n"; break;
  default: std::cout << "x␣*␣x␣>=␣4\n";
}
```

4.3.4 Атрибут [[fallthrough]]

Зачастую отсутствие `break` — скорее ошибка разработчика, нежели желаемое поведение. Существует не так много ситуаций, когда мы хотим выполнения сразу нескольких подряд идущих веток.

Поэтому компиляторы заботливо выдают предупреждения (ошибку при флаге *-Werror*), если встречают `case` без `break` :

```
main.cpp:10:26: error: this statement may fall through [-Werror=implicit-fallthrough=]
```

Чтобы успокоить компилятор и сказать, что мы так и хотели, можно дописать *атрибут* `[[fallthrough]]`

```cpp
switch (std::cin >> x; x * x) {
  case 0: [[fallthrough]]
  case 1: std::cout << "x␣*␣x␣<=␣1\n"; break;
  default: std::cout << "x␣*␣x␣>=␣4\n";
}
```

Теперь компилятор будет уверен, что вы знаете, что делаете в `case 0` .

5 Циклы

5.1 Цикл while

Цикл - оператор, позволяющий организовать повторяющееся выполнение другого оператора.
`while (<condition>) <statement>`

- `condition` — либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool`
- `statment` — оператор(statment) (может быть составной)

5.1.1 Решение простой задачи

Вводится n целых чисел, найти их сумму.

```cpp
#include <iostream>

int main() {
  int n;
  std::cin >> n;
  int sum = 0;
  while (n > 0) {
    int x;
    std::cin >> x;
    sum += x;
    --n;
    }
  std::cout << sum << '\n';
  return 0;
}
```

5.1.2 Примеры

Сами простыми примерам являются:

```cpp
// endless cycle
while (true) std::cout << 0;
```

```cpp
x = -5;
while (int sqr = x * x) {
  ++x;
  std::cout << sqr << '\n';
}
```

```cpp
// empty loop (spins while x is true)
while (x);
```

Замечание: последний цикл - *Undefined Behaviour*, если x не изменяет своего результата и не имеет побочных действий.

5.2 Цикл do-while

Цикл `do-while` аналогичен циклу `while` , за исключением того, что оператор(statment) цикла выполняется до проверки условия.

```cpp
do <statement> while (<condition>);
```

Таким образом, гарантируется, что цикл совершит хотя бы одну итерацию.

```
int x;
do {
  std::cin >> x;
  std::cout << x * x << '\n';
} while (x);
```

5.3 Цикл for

Цикл `for` имеет следующую структуру в C++:

```
for ([init]; [condition]; [expression]) <statement>
```

- `init` — либо выражение, либо объявление. Область действия объявленной сущности совпадает с областью оператора.
- `condition` — либо выражение, либо объявление переменной с инициализатором. В любом случае значение должно быть приводимо к `bool` .
- `expression` — произвольное выражение, выполняющееся в конце итерации
- `statment` — оператор, выполняющийся в цикле

5.3.1 Эквивалентность цикла for и while

Цикл `for`

```
for ([init]; [condition]; [expression]) <statement>
```

эквивалентен циклу `while` слудющего вида:

```
{
  [init];
  while ([condition]) { // while(true), if condition is empty
    <statement>
    [expression];
  }
}
```

но при этом гораздо лучше читаем, поэтому на практике чаще используется `for` .

5.3.2 Примеры

Простыми примерами применениями цикла `for` являются:

```
for (int i = 0; i < n; ++i) std::cout << i << '\n';
```

```
for (int i = 0; i < n; i += 2) {
  std::cout << i << '\n';
}
```

```
for (std::cin >> x; x != 0; std::cin >> x) {
  std::cout << x * x << '\n';
}
```

```
// endless cycle
for (;;) ...
```

```
// analogue of while
for (; x;) ...
```

```
for (int i = 0, j = 0; i < n && j < m; ++i, ++j) ...
```

5.4 Управляющие операторы

5.4.1 Оператор(statement) break

оператор(statement) `break` позволяет досрочно завершить **выполнение цикла**:

```
for(int i =0; i < n; ++i) {
  int x;
  std::cin >> x;
  if (x == 0) {
    std::cout << "Division␣by␣zero\n";
    break;
  }
  std::cout << y / x << '\n';
}
```

5.4.2 Оператор(statement) continue

Оператор(statement) `continue` позволяет досрочно завершить **текущую итерацию**:

```
for(int i =0; i < n; ++i) {
  int x;
  std::cin >> x;
  if (x == 0) {
    std::cout << "Division␣by␣zero\n";
    continue;
  }
  std::cout << y / x << '\n';
}
```

5.4.3 Оператор(statement) безусловного перехода

Оператор(statement) `goto` позволяет совершить "прыжок"в произвольное место функции, обозначенное некоторой "меткой".

```
// the program counts x and exits
int main() {
  int x;
  std::cin >> x;
  goto label;

  int y;
  std::cin >> y;
  std::cout << x + y << '\n';
  label:
  return 0;
}
```

Через `goto` может быть реализован цикл:

```
for (int i = 0; i < n; ++i) ...
// <=>
  int i = 0;
loop:
  if (i < n) {
    ...
    ++i;
    goto loop;
  }
```

Также оператор `goto` лежит в основе оператора `switch`, так как все случаи (`case` и `default`) являются метками для перехода. Поэтому даже следующая программа скомпилируется:

```
switch (x) {
  case 0: std::cout << "Zero\n"; break;
  case 1: std::cout << "One\n"; break;
  deault: std::cout << "Other\n"; // with a missing letter f
}
```

Оператор `goto` сильно усложняет чтение и отладку программ.

Во всех (даже безвыходных) ситуациях можно обойтись без него.