Лекция 1. Знакомство с языком C++

28. IDE

Создатель C++

C++ - компилируемый язык со статической типизацией.

Бьёрн Страуструп в 1979 году начал разработку языка.

Сайт Bjarne Stroustrup

Принцип нулевых накладных расходов, заложенный в основу языка C++, гласит:

1. Вы не платите за то, что не используете.
2. То, что вы используете, также эффективно, как если бы было написано вручную.

Только механизм RTTI и Исключений противоречат этому принципу.

Стандарты языка C++

Стандарт C++ — это официальный документ, специфицирующий язык программирования C++, который выпускается Международной организацией по стандартизации (ISO) и Международной электротехнической комиссией (IEC).

Стандарт определяет синтаксис языка, семантику выражений, состав стандартной библиотеки (STL).

Сайт Комитета по стандартизации C++

Версии стандарта:

- c++98
- c++03
- c++11
- c++14
- c++17
- c++20
- c++23 <=
- c++26

Плейлист Standard C++, YouTube Константин Владимиров

Компилятор

Популярные компиляторы C++:

- GNU Compiler Collection (GCC)
- Clang
- Microsoft Visual C++ (MSVC)
- Intel C++ Compiler

Поддержка компиляторами возможностей стандарта представлена по ссылке

Справка

- cppreference - онлайн-документация

Hello C++

Программа на C++ состоит из инструкций. Инструкция (statement) описывает выполнение определенного действия. Инструкция завершается символом `;`.

Блок кода - набор инструкций, заключенный между `{` и `}`.

Простейшая программа на C++ может состоять из одного файла.
Создадим файл с именем `main.cpp` и содержимым:

```
#include <iostream> // включение заголовочного файла ввод-вывод

// функция, с которой начинается исполнение программы
int main() {
    std::cout << "Hello and welcome to C++" << std::endl;
    return 0; // возвращаемое значение функции
}
```

Чтобы скомпилировать программу, необходимо на вход компилятору подать путь до файла.

```
g++ main.cpp
```

По умолчанию в Linux будет создан исполняемый файл `a.out`.
Запустить его, чтобы увидеть вывод программы, можно командой:

```
./a.out
```

Скомпилируем программу в исполняемый файл с именем `hello`

```
g++ main.cpp -o hello
```

Запустим программу:

```
./hello
```

Комментарии в коде

- однострочные комментарии `//`
- многострочные комментарии `/* ... */`

Типы данных

Для каждой переменной на этапе компиляции должен быть определен тип данных. Про каждый тип данных заранее известно сколько байт он занимает в памяти.

Классификация типов данных:

- Фундаментальные типы данных:
  - `void`
  - `std::nullptr_t`
  - интегральные типы
    - булевый тип (`bool`)
    - символьные типы (`char`, `signed char`, `unsigned char`, ...)
    - знаковые целочисленные типы (`short`, `int`, `long`, `long long`)
    - беззнаковые целочисленные типы (`unsigned short`, `unsigned int`, ...)
  - типы для чисел с плавающей точкой (`float`, `double`, `long double`)
- Составные типы данных
  - ссылочные типы
  - типы с указателем
  - типы с указателем на член класса
  - массивы
  - типы функций
  - перечисления
  - классы

Размер фундаментальных типов данных:

```
std::cout << "char: " << sizeof(char) << "\n";               //  1
std::cout << "bool: " << sizeof(bool) << "\n";               //  1
std::cout << "short: " << sizeof(short) << "\n";             //  2 (isocpp >= 2)
std::cout << "int: " << sizeof(int) << "\n";                 //  4 (isocpp >= 2)
std::cout << "long: " << sizeof(long) << "\n";               //  8 (isocpp >= 4)
std::cout << "long long: " << sizeof(long long) << "\n";     //  8 (isocpp >= 8)
std::cout << "float: " << sizeof(float) << "\n";             //  4
std::cout << "double: " << sizeof(double) << "\n";           //  8
std::cout << "long double: " << sizeof(long double) << "\n"; // 16
```

```
std::cout << "void: " << sizeof(void) << "\n"; //  compile-error or 1
std::cout << "std::nullptr_t: " << sizeof(nullptr) << "\n";  //  8
std::cout << "size_t: " << sizeof(size_t) << "\n";           //  8
```

- тип `size_t` предназначен для индексирования массивов и контейнеров. Стандартном гарантировано, что имеет размер, который вмещает все возможные адреса в памяти.

Типы данных фиксированного размера

Размер некоторых фундаментальных типов данных зависит от платформы для которой компилируется программа. Интегральные типы фиксированного размера представлены в заголовочном файле `<cstdint>`:

- знаковые: `int8_t`, `int16_t`, `int32_t`, `int64_t`
- беззнаковые: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

Как правило, данный заголовочный файл не приходится подключать, поскольку он включен в часто используемые фрагменты стандартной библиотеки.

Предельные значения различных типов

Предельные значения различных интегральных типов и типов с плавающей точкой можно получить с помощью шаблонного класса `std::numeric_limits` в заголовочном файле `<limits>`.

- для интегральных типов:

```
std::cout << "type\t| min()\t\t| max()" << '\n';
std::cout << "bool\t| "
          << std::numeric_limits<bool>::min() << "\t\t| "
          << std::numeric_limits<bool>::max() << '\n';
std::cout << "char\t| "
          << +std::numeric_limits<char>::min() << "\t\t| "
          << +std::numeric_limits<char>::max() << '\n';
std::cout << "uchar\t| "
          << +std::numeric_limits<unsigned char>::min() << "\t\t| "
          << +std::numeric_limits<unsigned char>::max() << '\n';
std::cout << "short\t| "
          << std::numeric_limits<short>::min() << "\t\t| "
          << std::numeric_limits<short>::max() << '\n';
std::cout << "int\t| "
          << std::numeric_limits<int>::min() << "\t| "
          << std::numeric_limits<int>::max() << '\n';
std::cout << "unsigned int\t| "
          << +std::numeric_limits<unsigned>::min() << "\t\t| "
          << +std::numeric_limits<unsigned>::max() << '\n';
std::cout << "long long\t| "
```

```
          << std::numeric_limits<long long>::min() << "\t| "
          << std::numeric_limits<long long>::max() << '\n';
std::cout << "size_t\t| "
          << std::numeric_limits<size_t>::min() << "\t| "
          << std::numeric_limits<size_t>::max() << '\n';
```

- для чисел с плавающей точкой:

```
std::cout << "type\t| lowest()\t| min()\t\t| max()" << '\n';
std::cout << "float\t| "
          << std::numeric_limits<float>::lowest() << "\t| "
          << std::numeric_limits<float>::min() << "\t| "
          << std::numeric_limits<float>::max() << '\n';
std::cout << "double\t| "
          << std::numeric_limits<double>::lowest() << "\t| "
          << std::numeric_limits<double>::min() << "\t| "
          << std::numeric_limits<double>::max() << '\n';
std::cout << "long double\t| "
          << std::numeric_limits<long double>::lowest() << "\t| "
          << std::numeric_limits<long double>::min() << "\t| "
          << std::numeric_limits<long double>::max() << '\n';
```

Переполнение

Превышение лимита приводит к переполнению.

Так для беззнаковых целочисленных типов это нормальная ситуация, применяется арифметика по модулю степени двойки (просто начнется новый отсчет).

Для знаковых целочисленных типов в соответствии со Стандартом это приводит к неопределенному поведению (***UB - undefined behavior***)

Для беззнаковых типов, следует быть внимательным при вычитании, так как можно получить большое значение.

Для чисел с плавающей точкой это валидная операция, которая приводит к значению бесконечность или денормализации числа.

```
int max_int = std::numeric_limits<int>::max();
std::cout << "MAX_INT = " << max_int << std::endl;
std::cout << "MAX_INT + 1 = " << max_int + 1 << std::endl; // UB

unsigned int max_uint = std::numeric_limits<unsigned int>::max();
std::cout << "MAX_UINT = " << max_uint << std::endl;
std::cout << "MAX_UINT + 1 = " << max_uint + 1 << std::endl; // wrap-around

float max_float = std::numeric_limits<float>::max();
std::cout << "MAX_FLOAT = " << max_float << std::endl;
std::cout << "MAX_FLOAT * 2 = " << max_float * 2.0f << std::endl; // inf
```

```cpp
float min_float = std::numeric_limits<float>::min();
std::cout << "MIN_FLOAT = " << max_float << std::endl;
std::cout << "MIN_FLOAT / 2 = " << min_float / 2.0f << std::endl; // subnormal
```

Идентификаторы

Идентификатор - имя переменной, функции, класса, перечисления, пространства имен и других сущностей языка.

Идентификатор представляет собой произвольной длины последовательность из:

- строчных `[a-z]` латинских символов;
- прописных `[A-Z]` латинских символов;
- символа подчеркивания `_` ;
- цифр `[0-9]` ;
- большинства символов Unicode (ограничено компиляторами).

В качестве первого символа идентификатора **НЕ** могут использоваться цифры `[0-9]` или символы Unicode **НЕ** имеющие свойства `XID_Start`

Сущностям C++ следует давать осмысленное имя (идентификатор).

One Definition Rule (ODR)

- каждая сущность в программе должна быть определена один раз.

Переменные

Переменная - именованная область памяти.

В C++ объявление переменной является определением. Для чистого объявления используется ключевое слово `extern` перед типом, которое означает, что переменная определена в другой единице трансляции (в другом файле).

Синтаксис объявления и определения переменной:
`<type> <name>;` , `<type> <name> = <value>;`

Синтаксис только объявления переменной: `extern <type> <name>;`

```cpp
char c = '1';
char ce = '\\';
bool b = true;
int i = 42;
unsigned ui = 42; // unsigned int
short int si = 17; // short
long li = 12321321312;
```

```
long long lli = 12321321312;
float f = 2.71828;
double d = 3.141592;
long double ld = 1e15;
```

Литералы

Литерал - константа времени компиляции. Значение, которое записывается в исходном коде программы.

Целочисленные литералы

Целочисленные литералы различных систем счисления:

- `42` - десятичная, литерал начинается с `[1-9]` и далее `[0-9]`
- `042` - восьмеричная, литерал начинается с `0` и далее `[0-7]`
- `0x2a`, `0X2A` - шестнадцатеричная, литерал начинается с `0x`, `0X` и далее `[0-9]`, `[a-f]`, `[A-F]`
- `0b101010` - двоичная (`C++14`), литерал начинается с `0b` и далее `[0-1]`

```
int dec = 42;
int oct = 042;
int hex = 0x42;
int bin = 0b101010;
```

Целочисленные литералы с суффиксами, обозначающими тип:

- `u`, `U` - суффикс для типа `unsigned`;
- `l`, `L` - суффикс для типа `long`;
- `ll`, `LL` - суффикс для типа `long long` (`C++11`);
- `z`, `Z` - суффикс для типа `size_t` (`C++23`);

Визуальное разделение цифр с помощью `'` (`C++14`):

- `18'123'000` - наглядно разделены тысячи числа.

```
int value = 18'123'000;
```

Литералы чисел с плавающей точкой

Литералы чисел с плавающей точкой, имеющие обозначающий тип суффикс:

- `f`, `F` - для типа `float`;

- `l`, `L` - для типа `long double`;

Литералы чисел с плавающей точкой:

- `.` - разделитель целой и дробной части, тип `double`
- `e` - экспоненциальная форма записи числа, тип `double`

Литерал с плавающей точкой в шестнадцатеричной системе счисления:

- `0x`, `0X` - в качестве префикса
- `p` - экспоненциальная форма записи

Символьные литералы

- `'a'`, `'\n'` - одиночные символы в кавычках и escape-последовательности;
- `u8` - префикс для символа UTF-8 (`C++17`);
- `u` - префикс для символа UTF-16 (`C++11`);
- `U` - префикс для символа UTF-32 (`C++11`);
- `L` - префикс для расширенного типа wchar_t;

Строковые литералы

- `""` - строка типа `const char[N]` (C-style)
- `R"(\n\n)"` - сырая строка, где символы escape-последовательности игнорируются
- `L""` - строка типа `const wchar_t[N]`
- `u8""` - строка символов UTF-8 типа `const char_t[N]` (until `C++20`), `const char8_t[N]` (since `C++20`)
- `u""` - строка символов UTF-16 типа `const char16_t[N]`
- `U""` - строка символов UTF-32 типа `const char32_t[N]`

Литералы логического типа

- `true` - для значения истина;
- `false` - для значения ложь.

Литерал для обозначения нулевого указателя

- `nullptr` - для обозначения нулевого указателя.

Пользовательские литералы

Начиная с `C++11`, в языке можно определять пользовательские литералы для различных типов.

Синтаксис: `<return_type> operator""_<lit>(<value_type> value) {<code>}`

- `<return_type>` - тип возвращаемого значения.
- `""` - обозначение оператора для суффиксного литерала.
- `_<lit>` - пользовательский суффиксный литерал.
- пробел между `""` и `_<lit>` отсутствует.
- `<value_type>` - тип принимаемого значения.

Ограничения:

- только суффиксный литерал, префиксный невозможно создать пользователю;
- начинается с `_`, без подчеркивания зарезервированы для стандартной библиотеки;
- ограничен возможными принимаемым типом данных:
  - `unsigned long long` - для целочисленных литералов;
  - `long double` - для чисел с плавающей точкой;
  - `const char*` - сырой литерал для вышеупомянутых типов;
  - `char`, `wchar_t`, `char8_t` (`C++20`), `char16_t`, `char32_t` для символов
  - `const <char_type>*, std::size_t` - для строковых литералов

Пример использования для различных единиц измерения:

```
// Литерал для расстояния в метрах
constexpr long double operator""_m(long double meters) {
    return meters;
}

// Литерал для расстояния в сантиметрах
constexpr long double operator""_cm(long double cm) {
    return cm / 100.0;
}

// Литерал для расстояния в киллометрах (целочисленный для примера)
constexpr unsigned long long operator""_km(unsigned long long km) {
    return km * 1000;
}

auto ivleeva = 13.0_cm;
auto height = 1.96_m;
auto to_the_moon = 384'400_km;
```

Автоматический вывод типа `auto`

Компиляторы C++ умеют самостоятельно выводить типы переменных по значению (`C++11`), которое им присваивается. Для этого вместо типа переменной используется ключевое слово `auto` (спецификатор типа);

```
auto b = false;   // bool

auto i = 42;      // int
auto u = 42u;     // unsigned int
auto ull = 42Ull; // unsigned long long
auto ihe = 0x1e5; // int
auto l = 42l;     // long
auto x = 9223'372'036'854'775'806; // long or long long

auto d = 3.14;    // double
auto dp = .1;     // double
auto de = 1e5;    // double
auto ds = 314e-2; // double
auto f = 3.14f;   // float
auto ld = 42.l;   // long double

auto c = 'a';     // char
auto u8c = u8'a'; // char
auto u16c = u'a'; // char16_t
auto u32c = U'a'; // char32_t
auto wc = L'a';   // wchar_t

auto s = "abc";   // const char*
auto rs = R"(popular escape-sequences in "C++" '\n' '\t')"; // const char*
```

Операторы

В языке представлены унарные, бинарные и тернарный операторы. Операторы представляют собой функции. Операторы выполняют действия над операндами и возвращают результат операции.

Общие операторы

Операнды `a`, `b`, `c`

- Инкремент/декремент: `++a`, `--a`, `a++`, `a--`
- Арифметические: `+a`, `-a`, `a + b`, `a - b`, `a * b`, `a / b`, `a % b`
- Побитовые: `~a`, `a & b`, `a | b`, `a ^ b`, `a << b`, `a >> b`
- Присваивание: `a = b`, `a += b`, `a -= b`, `a *= b`, `a /= b`, `a %= b`, `a &= b`, `a |= b`, `a ^= b`, `a <<= b`, `a >>= b`
- Логические: `!a`, `a && b`, `a || b`
- Сравнения: `a == b`, `a != b`, `a < b`, `a > b`, `a <= b`, `a >= b`, `a <=> b`

- Доступа к элементам: `a[...]`, `*a`, `&a`, `a->b`, `a.b`, `a->*b`, `a.*b`
- Другие: `a(...)`, `a, b`, `a ? b : c`

Специальные операторы

- `static_cast` - преобразует один тип в другой связанный тип
- `dynamic_cast` - преобразует внутри иерархий наследования
- `const_cast` - добавляет или удаляет cv-квалификаторы
- `reinterpret_cast` - преобразует один тип в другой несвязанный тип
- `(type)value` - C-style cast
- `new` - создает объекты с динамическим временем жизни
- `delete` - разрушает объекты, ранее созданные выражением `new`,
  и освобождает выделенную область памяти
- `sizeof` - запрашивает размер типа
- `sizeof...` - запрашивает размер пачки параметров (начиная с C++11)
- `typeid` - запрашивает информацию о типе
- `noexcept` - проверяет, может ли выражение вызвать исключение (начиная с C++11)
- `alignof` - запрашивает требования к выравниванию типа (начиная с C++11)

Приоритет операторов

1. -> : `a::b`
2. -> : `a++`, `a--`, `<type>(a)`, `<type>{a}`, `a()`, `a[]`, `a.b`, `a->b`
3. <- : `++a`, `--a`, `+a`, `-a`, `!a`, `(<type>)a`, `*a`, `&a`, `sizeof`,
   `new`, `delete`
4. -> : `a.*b`, `a->*b`
5. -> : `a * b`, `a / b`, `a % b`
6. -> : `a + b`, `a - b`
7. -> : `a << b`, `a >> b`
8. -> : `a <=> b` (`C++20`)
9. -> : `a < b`, `a <= b`, `a > b`, `a >= b`
10. -> : `a == b`, `a != b`
11. -> : `a & b`
12. -> : `a ^ b`
13. -> : `a | b`
14. -> : `a && b`
15. -> : `a || b`
16. <- : `a ? b : c`, `throw`, `a = b`, `a += b`, `a -= b`, `a *= b`, `a /= b`,
    `a %= b`, `a <<= b`, `a >>= b`, `a &= b`, `a ^= b`, `a |= b`
17. -> : `a, b`

Приведение типов

- неявное приведение типов (***implicit***)
- явное приведение типов (***explicit***)

Неявное приведение арифметических типов

Неявное приведение типов происходит когда типы операндов отличаются друг от друга, тип принимаемого аргумента функции отличается.

Промотирование типа (***promotion***):

- все интегральные типы, имеющие размер меньше `int` (`char`, `short`), продвигаются к `int`.
- вещественный тип `float`, продвигается к `double`.

Преобразование интегральных типов

- для знаковых: `int` -> `long` -> `long long` (к большему размеру)
- для беззнаковых: `unsigned int` -> `unsigned long` -> `unsigned long long`
- если один знаковый, а другой беззнаковый, то преобразуется к беззнаковому по модулю 2^n
- сужающие преобразования возможны, но могут происходить с потерей точности.

Преобразование вещественных типов

- преобразование `float` -> `double` -> `long double`
- сужающие преобразования возможны, но могут происходить с потерей точности.

Преобразование из вещественного в интегральный тип

- происходит отбрасывание вещественной части.

```
int i = -43.9;     // -43
unsigned u = 24.9; // 24
```

Преобразование в булевый тип:

- `false` - 0 для интегральных типов, 0.0 для чисел с плавающей точкой, `nullptr` для указателей.
- `true` - для всех остальных значений.

```
bool b_int = -42;       // true
bool b_double = 1e-308; // true
bool b_uint = 42u;      // true
```

```
bool b_char = '0';      // true '0' is not '\0'
bool b_null = nullptr;  // false bool b_d_zero = 0.0;    // false
```

Преобразование из булевого типа:

- `false` - значение 0 (0.0)
- `true` - значение 1 (1.0)

```
int i_true = true;      // 1
double d_false = false; // 0
```

Явное приведение арифметических типов

В коде явно указывается тип к которому необходимо выполнить преобразование.

- `(type)<var>` - C-style cast.
- `type(<expr>)` - функциональная нотация.
- `static_cast<type>(<expr>)` - явное приведение к типу, указанному в `<>`.

В C++ следует использовать `static_cast`, поскольку это безопасное приведение типа, так как на этапе компиляции проверяется возможность преобразования.

```
double d = 3.14159;
int i = -42;
char c = 'D';

int i = (int)d; // C-style cast
unsigned u = unsigned(i); // functional notation
float f = static_cast<float>(c); // static_cast C++
```

Выражения (expression)

Выражение - это последовательность операторов и их операндов, которые задают вычисления.

Изменить приоритет выполнения операторов в выражении возможно с помощью `()`.

Особенности арифметических операций

- Оператор `/` при делении целочисленных величин, производит деление нацело и дробная часть отбрасывается.

```
auto res = 5 / 2; // res = 2;
double d = 5 / 2; // d = 2.0;
```

Для получения дробной части, необходимо чтобы один из операндов имел тип с плавающей точкой:

```
auto res = 5. / 2; // double res = 2.5;
double d = static_cast<double>(5) / 2; // d = 2.5
```

- Оператор `%` позволяет получить остаток от целочисленного деления:

```
auto res = 5 % 2; // res = 1;
```

Логические операции

- `!a` - инверсия
- `a && b` - логическое И (конъюнкция)
- `a || b` - логическое ИЛИ (дизъюнкция)

Возможно собирать цепочку из логических операций.

short-circuit evaluation

- вычисление цепочки конъюнкций заканчивается при первом появлении `false`
- вычисление цепочки дизъюнкций заканчивается при первом появлении `true`

```
bool a = true, b = false, c = false;

bool res_and = a && b && c; // a && b
bool res_or = a || b || c; // a
```

Начиная с `C++17` данное свойство сохраняется при переопределении операторов.

Битовая арифметика

Может пригодиться для хранения битовой маски или более компактной упаковки нескольких маленьких переменных в одну.

- `x = x | (1u << n)` - установить бит под номером `n` в числе `x` в значение 1
- `x = x & ~(1u << n)` - установить бит под номером `n` в числе `x` в значение 0
- `b = (x >> n) & 1u` - считать значение бита под номером `n` в числе `x` (`b = x & (1u << n)`)
- `x = x ^ (1u << n)` - инвертировать бит под номером `n` в числе `x`

- `x = x ^ x` - преобразовать все биты в значение 0
- `x = x ^ ~x` - преобразовать все биты в значение 1

Тернарный оператор `?:`

Синтаксис: `<condition> ? <expression_when_true> : <expression_when_false>`

Как и любой оператор тернарный оператор возвращает определенный тип. Поэтому результат выражений, возвращаемый при выполнении истинного и ложного условия, должен быть идентичным.

```
bool condition = false;
condition ? std::cout << "is True\n" : std::cout << "is False\n";
```

- если в одном из условий заменить `std::cout` на `std::cerr`, то код будет работать, так как оба объекта имеют одинаковый тип `std::ostream`.

Если возвращаемые типы выражений разные, то будет ошибка.

```
bool condition = false;
std::string str = "";
condition ? std::cout << "is True\n" : str = "is False\n"; // compilation error
```

- но данную логику можно сохранить, если обернуть данные фрагменты кода в функцию, например возвращающую тип `void` или использовать оператор запятая.

Оператор `,`

У символа `,` есть много назначений и далеко не везде в коде символ `,` является оператором. Оператор запятая встречается в выражениях.

```
int n = 1;
int m = (++n, std::cout << "n = " << n << '\n', ++n, 2 * n);
std::cout << "n = " << (++n, n) << '\n';
std::cout << "m = " << m << std::endl;
```

Вывод:

```
n = 2
n = 4
m = 6
```

Область видимости

В языке C++ можно выделить следующие области видимости (***scope***):

- ***global scope*** - глобальная область видимости охватывает всю программу.
- ***namespace scope*** - область видимости пространства имен.
- ***block scope*** - область видимости блока кода.
- ***enumeration scope*** - область видимости перечисления.
- ***function parameter scope*** - область видимости параметров функции.
- ***class scope*** - область видимости класса.
- ***lambda scope*** - область видимости лямбда-выражения.
- ***template parameter scope*** - область видимости шаблонных параметров.

Область видимости определяет фрагмент кода программы, в котором объявленная сущность C++ (переменная, функция, класс, ...) является видимой и доступной для использования.

Сущность вводится в программу посредством объявления. Место объявления определяет начало области видимости.

В одной области видимости нельзя определить переменную с тем же именем. Можно объявлять переменные в более локальной (вложенной) области видимости с тем же именем. Для других сущностей C++ логика аналогичная, но есть отличия. Например, для функций можно определять функцию с тем же именем, но с другими аргументами (перегрузка функций).

Во вложенной области видимости видны переменные и сущности из внешней области. Переменные и сущности из внутренней области видимости не видны во внешней.

Глобальная область видимости

Охватывает всю программу.

В глобальной области видимости происходит инициализация переменных значениями по умолчанию (нулевыми значениями).

Область видимости блока кода

Блок кода заключенный в `{<code>}` создает область видимости.

К области видимости блока кода относятся также ветвления, циклы, обработчик исключений.

Появление переменной в области видимости происходит в момент `<type> <name>` до `=`, поэтому следующий код приводит к UB (undefined behavior - неопределенное поведение):

```
int x = 42;
{
```

```
    int x = x; // UB, x != 42, because inner x is in scope before `= x`
}
int y = x; // y = 42
```

Сокрытие переменной (shadowing)

При объявлении переменной во вложенной области видимости с тем же именем, что и во внешней, происходит сокрытие видимости внешней переменной, она становится не доступна для локальной области видимости.

```
int x = 42;
{
    x = 33; // update value of x from outer scope
    int x = 24; // shadowing
    std::cout << "x = " << x << std::endl; // x = 24
} // end of local x
std::cout << "x = " << x << std::endl; // x = 33
```

Пространство имен

Пространство служит для организации кода в именованные блоки, что позволяет избежать конфликта имен и лучше структурировать программу. Пространство имен гарантирует уникальность имен во внутренней области кода.

Синтаксис: `namespace <name> {<code>}`

Принято после `}` добавлять комментарий `// namespace <name>`, указывающий какое пространство имен закрывает соответствующая фигурная скобка.

```
namespace phasor {
    // переменные, функции, классы, перечисления и другие сущности
} // namespace phasor
```

Пространство имен можно открывать и закрывать любое количество раз, дописывая внутри пространства имен новые сущности.

Вложенные пространства имен объявляются внутри другого пространства имен:

```
namespace A {
namespace B {
    // переменные, функции, классы, перечисления и другие сущности
} // namespace B
} // namespace A
```

Оператор `::`

Указание пространства имен осуществляется посредством оператора разрешения области видимости `::`.

Глобальное пространство имен не имеет имени и начинается с оператора `::`. Как правило, указание глобального пространства имен можно опустить.

Доступ к глобальной переменной при наличии одноименной переменной в области видимости функции осуществляется с помощью `::`:

```
int val = 10;

int main() {
    int val = 20;
    std::cout << "local val = " << val << std::endl;
    std::cout << "global val =" << ::val << std::endl;
}
```

В `C++17` можно открывать вложенные пространства имен, используя `::`:

```
namespace A::B {
    // переменные, функции, классы, перечисления и другие сущности
} // namespace B
```

Безымянное пространство имен

Безымянное пространство имен создает область видимости уникальную для каждой единицы трансляции.

Синтаксис: `namespace {<code>}`

Объявленные внутри данного пространства имен будут иметь внутреннюю компоновку (internal linkage).

Особенности:

- Эквивалентно ключевому слову `static` для функций и переменных.
- Данное пространство имен невозможно открыть в другом файле и дополнить его.
- Позволяет скрывать реализацию. Элементы из данного пространства не видны вне данного файла.
- Уникальное имя пространства предотвращает конфликт имен.

`using namespace`

Подключение пространства имен осуществляется с помощью синтаксиса `using namespace <identifier>`, что включает все сущности из пространства имен в текущую область видимости. После чего все сущности доступны без указания пространства имен.

Не рекомендуется использовать в глобальной области видимости.

`using`

Подключение определенной сущности из пространства имен осуществляется с помощью `using`, после которого указывается сущность.

```
using std::cout, std::cin;
using std::literals::string_literals;
```

- как правило, лучше писать внутри функции, где непосредственно используется сущность, тем самым уменьшая область видимости.

Объявление псевдонимов (алиасов)

Использование `typedef` является наследием языка C:

```
typedef std::string str_t;
```

В современном C++ предпочтительнее использовать `using`:

```
using str_t = std::string;
```

Массивы

Массив расположен в памяти линейно.

Синтаксис: `<type> <name>[<size>] = {<elem_0>, <elem_1>, ...};`

Можно не указывать `<size>`, тогда создастся массив по количеству элементов соответствующий инициализирующим значениям в `{}`.

Доступ к элементу массива через оператор `[]`.

Инициализация массива

Инициализация массива:

```
int arr1[5];                    // неинициализированный массив
int arr2[5] = {1, 2, 3, 4, 5}; // явная инициализация
int arr3[] = {1, 2, 3, 4, 5};  // автоматическое определение размера
int arr4[5] = {1, 2};          // остальные элементы = 0
int arr5[5] = {};              // все элементы = 0
```

Новые способы инициализация в `C++11` (uniform initialization):

```
int arr3[]{1, 2, 3, 4, 5}; // uniform initialization
int arr4[5]{1, 2, 3};  // [1, 2, 3, 0, 0] partial initialization
int arr5[10]{};
```

Не используется с ключевым словом `auto`:

```
auto arr = {1, 2, 3, 4, 5};  // type is std::initializer_list<int>
```

Массив символов:

```
char str_one[] = "Hello"   " "   "bro";
char str_two[] = {'H', 'e', 'l', 'l', 'o', ' ', 'b', 'r', 'o', '\0'};
```

- причем `str_one` посимвольно равен `str_two`.

Массив булевых элементов по умолчанию инициализируется `0` - `false`.

Определение количество элементов массива

Под размером контейнера или массива в C++ подразумевается количество элементов.

Оператор `sizeof` возвращает размер массива в байтах. Поэтому для определения размера необходимо знать тип элементов и, следовательно, размер этого типа.

Синтаксис: `sizeof(<array>) / sizeof(<type>)`
Или: `sizeof(<array>) / sizeof(<array>[0])`

```
int array[] = {2, 4, 5, 6, 7};
size_t size = sizeof(array) / sizeof(int);

char word[] = "word";
size_t size = sizeof(name) / sizeof(word[0]); // 5
```

Другой способ явно указать размер массива при создании, используя константное значение или `constexpr` выражение (C++11):

```
const int size = 10;
char str[size];
```

```
constexpr int count = 20;
double result[count];
```

Начиная с `C++17` существует специальная шаблонная функция `std::size` в заголовочном файле `<array>` (и других заголовочных файлах):

```
int array[] = {2, 4, 5, 6, 7};
size_t arr_size = std::size(array);
```

Двумерные массивы

Двумерные массивы также расположены в памяти линейно.

Синтаксис: `<type> <name>[<size_x>][<size_y>] = { {<x_0>, ...}, {<y_0>, ...} };`

Инициализация:

```
int matrix1[3][3];                          // неинициализированный массив 3x3
int matrix2[2][3] = {{1, 2, 3}, {4, 5, 6}};   // явная инициализация
int matrix3[][3] = {{1, 2, 3}, {4, 5, 6}};    // автоопределение строк
int matrix4[3][3] = {{1}, {4, 5}, {7, 8, 9}}; // частичная инициализация
int matrix5[3][3] = {};                       // инициализация нулями
int matrix6[3][3] = {1, 2, 3, 4, 5, 6};       // плоская инициализация
```

Аналогично с `C++11` доступна uniform initialization.

Перечисления `enum` и `enum class`

Перечисления - специальный тип данных, который состоит из набора именованных констант. По умолчанию данный тип является типом `int` для `enum class` и *implementation-difened* для `enum`.

`enum`

- наследие языка C.
- именованные константы не имеют собственной области видимости, а попадают во внешнюю по отношению к перечислению область видимости
- неявно приводится к типу `int`

```
enum CommandType {
    CT_INIT,
    CT_UPDATE,
```

```
    CT_FINISH
};
```

`enum class` (C++11)

- именованные константы находятся в области видимости данного перечисления
- приведение к типу `int` только явное, неявно не приводится

```
enum class CommandType {
    INIT,
    UPDATE,
    FINISH
};
```

Указание базового типа

Возможно указать базовый интегральный тип данных, лежащий в основе перечисления.
Синтаксис: `enum [class] <name> : <type> {<enum>}`

Указание определенного значения

Можно задать произвольное значение для соответствующей именованной константы.

```
enum class Color {
    CYAN = 1,  // 1
    MAGENTA,   // 2
    YELLOW,    // 3
    BLACK = 0  // 0
};
```

Использование в качестве битовой маски:

```
enum ChecksFlag : uint8_t {
    CF_NONE = (0 << 0), // 0
    CF_TIME = (1 << 0), // 1
    CF_KEYS = (1 << 1), // 2
    CF_USER = (1 << 2), // 4
    CF_CERT = (1 << 3), // 8
    CF_ALL = CF_TIME | CF_KEYS | CF_USER | CF_CERT // 15
};
```

Можно создать `enum` с одинаковыми значениями (***лучше так не делать***):

```
enum class Command {
    INIT = 1,
    START = 2,
    PAUSE = 1,
    RESET = 3
};
```

- оператор `if` отработает в двух случаях для `Command::INIT`, `Command::PAUSE`
- оператор `switch` выдаст ошибку компиляции при попытке создать данные `case`

Можно задать значение для произвольного `enum`, тогда следующий элемент продолжит заданную нумерацию:

```
enum class Command {
    INIT = 1,   // 1
    START,      // 2
    PAUSE = 10, // 10
    RESET       // 11
};
```

Данный подход содержит опасность, так как в определенных случаях нумерация может пересечься, например при дописывании программы:

```
enum class Color {
    CYAN = 1,  // 1
    MAGENTA,   // 2
    YELLOW,    // 3
    BLACK = 0, // 0
    WHITE      // 1 !!! is duplicate
};
```

Определение количества элементов

Когда нумерация выполняется по умолчанию, с нуля, в конец перечисления можно добавить константу с именем `COUNT`, которая будет иметь номер соответствующий количеству элементов:

```
enum class CommandType {
    INIT,
    UPDATE,
    FINISH,
    COUNT
};
```

Преобразование несуществующего значения в перечисление

Возможно преобразовать значение к значению типа `enum`, которого нет в перечислении:

```
enum class ABC {
    A = 1,
    B = 3,
    C = 5
};

auto result = static_cast<ABC>(2);
std::cout << typeid(result).name() << std::endl; // 3ABC
```

Ветвление

Инструкция `if`

Синтаксис: `if (<condition>) {<code>}`

- `<condition>` - в качестве условия может использоваться любое логическое выражение или значение типа, неявно конвертируемого в bool (указатель, `int`)

```
if (<condition>) {
    // исполняемый код для случая истинного условия
}
```

Синтаксис с `else`:

```
if (<condition_1>) {
    // исполняемый код для случая истинного условия 1
} else if (<condition_2>) {
    // исполняемый код для случая истинного условия 2
} else {
    // исполняемый код на случай невыполнения условий выше
}
```

Возможность инициализации внутри `if` добавлена в `C++17`:

```
if (<init-statement>; <condition>) {
    // исполняемый код для случая истинного условия
}
```

Инструкция `switch`

Передает управление одной из нескольких наборов инструкций в зависимости от значения выражения в условии.

Используется с символьными, целочисленными типами данных, перечислениями.

Синтаксис: `switch (<condition>) {<code>}`

- `<condition>` - символьный, целочисленный тип данных, перечисления

В блоке кода используется метки `case` для определенного значения переменной и метка `default` для случаев, когда значение в блоке условия не совпало с метками `case`

```
switch (<condition>) {
    case <value_1>:
        // некоторый код для случая <condition> == <value_1>
        break;
    case <value_2>:
        // некоторый код для случая <condition> == <value_2>
    case <value_3>:
        // некоторый код для случая <condition> == <value_2> || <value_3>
        break;
    default:
        // некоторый код для случая <condition> != <value_#>
}
```

Особенности:

- Лучше применять для дискретных значений, нежели сложных условий.
- Возможность выполнения проверки за O(1) с jump table (оптимизация компилятора). Работает при плотных значениях, достаточном количестве условий (от 3-5), при отсутствии или простом `default`, при известных значениях на этапе компиляции.

Аналогично `if` в `C++17` добавлена возможность инициализации внутри `switch`.

Атрибут `[[fallthrough]]` - для того, чтобы показать компилятору, что проваливание в следующее условие `case` задумано разработчиком.

В случае использования перечислений компилятор может выводить предупреждения, если условия `case` существуют не для всех возможных значений перечисления.

```
switch (2) {
    case 2:
        int x = 0; // initialization
        std::cout << x << '\n';
        break;
    default:
        // compilation error: jump to default:
        // would enter the scope of 'x' without initializing it
        std::cout << "default\n";
        break;
}
```

Решение данной проблемы:

```
switch (2) {
    case 2: {
        int x = 0; // initialization
        std::cout << x << '\n';
        break;
    } // end of scope 'x'
    default:
        std::cout << "default\n";
        break;
}
```

Инструкция `goto`

Синтаксис `goto <label>;`
Безусловная передача управления на инструкции после метки.

Никогда не стоит пользоваться данным оператором. Доказано, что любой код можно переписать без данного оператора.

```
<label> :
{
    if (<condition>) {
        <statement>;
        goto <label>;
     }
}
```

Циклы

Цикл `while`

Цикл с предусловием.

Синтаксис: `while (<condition>) {<body>}`

```
int value = 0, sum = 0;
while (std::cin >> value) {
    sum += value;
}
```

- оператор `>>` после выполнения операции считывания, возвращает поток `std::cin`, а поток неявно преобразовывается в bool, пока нет ошибок и поток не закончился он имеет

значение `true`.

Цикл `do-while`

Цикл с постусловием. Сначала исполняется тело цикла, затем осуществляется проверка.

Синтаксис: `do {<body>} while (<condition>);`

Использование при обработке ошибок:

```
int error = 0;
do {
    // некоторый код, который выполняется и обновляет error
    if (error) {
        // обработка ошибки
        break;
    }
    // некоторый код, который выполняется и обновляет error
    if (error) break;
    // некоторый код, который выполняется
    if (error) break;
} while (false);
```

Использование при считывании значения с консоли или от другой функции.

Цикл `for`

Синтаксис: `for (<initialization>; <condition>; <expression>) {<body>}`

Пример:

```
for (int i = 0; i < 10; ++i) {
    std::cout << i << '\t' << i * i << '\n';
}
```

Действия после итерации цикла могут быть перечисленны с помощью оператора `,`:

```
for (int i = 0, j = 0; i < 10; ++i, j += 2) {
    std::cout << i << " * " << j << " = "  << i * j << '\n';
}
```

Вложенные циклы:

```
const size_t size = 3;
int matrix[size][] = {
    {1, 2, 3},
```

```
    {4, 5, 6},
    {7, 8, 9}
};
for (size_t i = 0; i < size; ++i) {
    for (size_t j = size - 1; j < size; --j) {
        std::cout << matrix[i][j] << '\n';
    }
}
```

Цикл range-based `for` (`C++11`)

Позволяет итерироваться по массиву, контейнеру.

Синтаксис: `for (<type> <identifer> : <container>) {<body>}`

```
char str[] = "separating";
for (auto c : str) {
    std::cout << c << " ";
}
std::cout << std::endl;
```

Начиная с `C++20` можно добавить инициализирующее выражение.
Синтаксис: `for (<init-statement>; <type> <identifer> : <container>) {<body>}`

Инструкции `break`, `continue`

- `break` - выход из цикла, как правило, используется при выполнении условия `if`
- `continue` - переход на следующую итерацию, в конец блока кода цикла, перед `}`

Пример с `break`:

```
char break_symbol = 'k';
for (char c = 'a'; c < 'Z'; c += 2) {
    if (c >= break_symbol) {
        break;
    }
    std::cout << i << " ";
}
std::cout << std::endl;
```

Пример с `continue`:

```
for (int i = 0; i < 10; ++i) {
    if (i % 2 == 0) {
        continue;
    }
```

```
    std::cout << i << " ";
}
std::cout << std::endl;
```

Бесконечные циклы

Бесконечный цикл `for`:

```
for (;;) {
    std::cout << "infinite loop" << '\n';
}
```

Бесконечный цикл `while`:

```
while (true) {
    bool is_finished = false;
    // исполняемый код, изменяющий переменную is_finished
    if (is_finished) {
        std::cout << "infinite loop is finished" << '\n';
        break;
    }
}
```

Функции

Функция - сущность языка C++, которая представляет собой именованный набор инструкций со списком входных параметров и типом возвращаемого значения.

Каждая функция имеет тип, включающая как тип возвращаемого значения, так и типы входных аргументов.

Синтаксис: `<return_type> <function_name>(<arguments[0, 1, ...]>) {<body>}`

- `<return_type>` - может быть `auto`.
- `<arguments[0, 1, ...]` - представляет собой произвольное число аргументов, каждый из которых имеет произвольный тип и идентификатор (имя), причем имя переменной может быть опущено.

Если функция ничего не возвращает, то указывается тип `void`.

Если функция не принимает аргументы, то `()` остаются пустыми.

Компилятор прямолинейный, поэтому при использовании функции (имени функции), он должен знать, что это за идентификатор (имя). Различают объявление и определение функции.

Объявление (declaration):

```
int Sum(int lhs, int rhs); // type: int(int, int)
```

- может быть произвольное количество объявлений.

Определение (definition):

```
int Sum(int lhs, int rhs) {
    return lhs + rhs;
}
```

- определение функции является также объявлением.
- единственное определение в программе.
- может совпадать по имени, но отличаться по количеству аргументов и/или их типу.

Вызов функции следующим синтаксисом: `<function_name>(<arguments[0, 1, ...]>)`

- внутри `()` - передаются нужные аргументы через `,`.

При вызове функции в качестве аргумента может быть передано возвращаемое значение другой функции. Нет никакой гарантии в каком порядке будут вызываться данные функции, вычисляться аргументы и передаваться.

```
int PrintValue(int value) {
    std::cout << "value = " << value << std::endl;
    return value;
}

void Nothing(int, int, int) {}

int main() {
    Nothing(PrintValue(1), PrintValue(2), PrintValue(3));
    int value = PrintValue(1) + PrintValue(2) * PrintValue(3) + PrintValue(4);
    std::cout << "value = " << value << std::endl;
}
```

Скомпилируем код с помощью GCC и запустим код:

```
  g++ main.cpp && ./a.out
```

Скомпилируем код с помощью Clang и запустим код:

```
  clang++ main.cpp && ./a.out
```

Порядок вывода значений должен оказаться разным.

Значение имени функции `__func__`

Начиная с `C++11` в языке появился идентификатор `__func__`, который хранит имя данной функции и доступен внутри тела функции.

Область видимости данного идентификатора начинается непосредственно перед телом функции.

```
void print_self_name() {
    std::cout << __func__ << std::endl; // print_self_name
}
```

Инструкция `return`

Возвращает значение выражения указанного после ключевого слова `return`.
Синтаксис `return <expression>;`

Тип возвращаемого значения должен совпадать с типом возвращаемого значения или неявно преобразовываться в него.

Функция может иметь несколько `return` в зависимости от написанной логики. Если функция возвращает `void`, то наличие `return` не обязательно.

Позволяет досрочно выйти из функции:

```
void Print(int[] arr, size_t size) {
    if (size == 0) {
        return;
    }
    // код программы
}
```

Сконструировать значение по умолчанию посредством `{}`:

```
std::string Print(int[] arr, size_t size) {
    if (size == 0) {
        return {};
    }
    // код программы
}
```

Аргументы по умолчанию

У аргументов функции можно указать значение по умолчанию после знака `=`. Аргументы со значением по умолчанию должны идти подряд справа налево:

```
void PrintValues(int x, float f = 0.0, char c = 'a') {
    std::cout << "x = " << x << ", f = " << f << ", c = " << c << std::endl;
}
```

Доступен вызов функции:

```
PrintValues(x, f, c);
PrintValues(x, f);
PrintValues(x);
```

В качестве параметров по умолчанию возможно использовать выражения, вызов других функции возвращающий соответствующий тип.

Область видимости функции

Имя функции находится во внешней области видимости.

Аргументы имеет независимую область видимости, которая распространяется на тело функции до `}` при определении, и до закрытия списка аргументов `)` при объявлении.

Это значит, что **нельзя**:

- использовать имя параметра для значения по умолчанию для другого параметра.
- определять внутри тела функции переменную с тем же именем, что и параметр.

Неинициализированные переменные внутри функции содержат мусор (произвольные значения) в отличие от глобальных переменных.

Передача массива в функцию

Если в функцию в качестве аргумента передается массив, то фактически передается указатель на первый элемент массива, происходит встроенное в язык C++ преобразование (array-to-pointer conversion).

Следовательно, внутри функции теперь нет возможности использовать range-based for, поскольку он не работает для указателей. Кроме того, не получится определить размер массива привычным способом, `size_t size = sizeof(arr) / sizeof(arr[0])` или `size_t size = std::size(arr)`;

Для массива символов всегда можно в конец массива добавить специальный символ завершения строки `\0` и тогда можно внутри функции работать с массивом в цикле:

```
void DoSomething(char chars[]) {
    for (size_t i = 0; chars[i] != '\0'; ++i) {
        // do something
```

```
    }
}
```

Для других типов одним из решений данной проблемы является передача размера массива аргументом:

```
void Print(int numbers[], size_t n) {
    for(size_t i = 0; i < n; ++i) {
        std::cout << numbers[i];
    }
    std::cout << std::endl;
}
```

Рекурсия

Рекурсивная функция - функция, которая вызывает сама себя.

Определим значение элемента последовательности Фибоначчи под номером `n`:

```
uint64_t FibonacciRecursive(uint64_t pos) {
    if (pos < 2) {
        return 0;
    }
    if (pos == 2) {
        return 1;
    }
    return FibonacciRecursive(pos - 1) * FibonacciRecursive(pos - 2);
}
```

Любую функцию можно переписать без рекурсии:

```
uint64_t Fibonacci(uint64_t pos) {
    if (pos < 2) {
        return 0;
    }
    int prev = 0;
    int result = 1;
    for (uint64_t i = 2; i < pos; ++i) {
        result += prev;
        prev = result;
    }
    return result;
}
```

Ввод-вывод `<iostream>`

В заголовочном файле `<iostream>` при подключении создаются объекты связанные со стандартными потока *stdin*, *stdout*, *stderr*:

- `std::cin` - стандартный поток ввода;
- `std::cout` - стандартный поток вывода;
- `std::cerr` - стандартный поток ошибок (небуферизированный);
- `std::clog` - стандартный поток логирования (буферизированный);

Для данных потоков переопределены соответствующие операторы побитового сдвига `>>` (для `std::cin`) `<<` (для остальных).

Поэтому их иногда называют оператор ввода из потока `>>` и оператор вывода в поток `<<`.

```
int age;
double budget;

std::cout << "Введите ваш возраст: ";
std::cin >> age; // пользователь вводит с клавиатуры

std::cout << "Введите ваше бюджет: ";
std::cin >> budget; // пользователь вводит с клавиатуры
```

Вывод и ввод нескольких значений

Как уже обсуждалось операторы это функции и у них есть возвращаемое значение. Данные операторы принимают два аргумента: поток и значение, которое нужно либо считать из потока, либо вывести в поток в зависимости от оператора.

Данный оператор имеет левую ассоциативность, то есть выполняется слева направо. Данные операторы возвращают поток, который они принимают в левым аргументом (первым), поэтому возможно использовать цепочку операторов для ввода и вывода.

```
int age;
double budget;

std::cin >> age >> budget;
std::cout << "Возраст = " << age << ", бюджет = " << budget << std::endl;
```

Различия между `std::endl` и `'\n'`

- `std::endl` - добавляет символ `\n` в буфер и сбрасывает буфер вывода `std::flush`
- `\n` - добавляет символ `\n` в буфер и **НЕ** сбрасывает буфер вывода.

За счет этого `\n` будет быстрее работать, чем `std::endl` так как буфер будет сбрасываться реже, по накоплению достаточно большого количества символов.

В случае критического завершения программы, данные записанные в буфер и не выведенные на консоль можно потерять. Поэтому при отладке рекомендуется использовать `std::endl` или использовать небуфферизированный поток ошибок `std::cerr` для которого не имеет значение `\n` или `std::endl`.

Сбросить буфер можно и без перевода на новую строку. Для этого в поток можно вывести манипулятор `std::flush` и вызвать соответствующий метод объекта потока `std::cout.flush()`.

Ускорение работы ввода-вывода

Часто для ускорения работы ввода-вывода при сдаче задач в соревнованиях по алгоритмам используют следующие инструкции:

1. Отключение синхронизации C++ потоков (*iostream*) с C-потоками (*stdio*):

```
std::ios_base::sync_with_stdio(false);
```

- По умолчанию C++ обеспечивает совместимость с C, поэтому `std::cout` и `printf`, `std::cin` и `scanf` могут смешиваться, что требует накладных расходов на синхронизацию.

2. Отвязывания потока ввода `std::cin` от `std::cout`:

```
std::cin.tie(nullptr);
```

- По умолчанию, перед каждым чтением из `std::cin` происходит сброс буфера `std::cout`. Это нужно в интерактивных программах, чтобы подсказки для пользователя выводились сразу.
- Можно аналогичной командой отвязать `std::cout` от других потоков, но, как правило, он не привязан к другим потокам.

Восстановить синхронизацию можно инструкцией: `std::ios_base::sync_with_stdio(true);`.

Привязать поток обратно можно с помощью `std::cin.tie(&std::cout);`.
Также можно сохранить предыдущую привязку перед отвязыванием:

```
auto original_tie = std::cin.tie(nullptr);
std::cin.tie(original_tie);
```

Работа с файлами ввода-вывода через консоль

Подать входные данные из файла на входной поток (*stdin*):

```
./a.out < input.txt
```

Вывод *stdout* в файл:

```
./a.out > output.txt
```

Вывод stderr в файл:

```
./a.out 2> errors.txt
```

Ввод из файла, *stdout* в *output.txt*, *stderr* в *errors.txt*:

```
./a.out < input.txt > output.txt 2> errors.txt
```

Склеить *stdout* и *stderr* в один файл:

```
./a.out > output_and_errors.txt 2>&1
```

```
./a.out >& output_and_errors.txt
```

Функция `main`

Функция `main` - точка входа в программу. Должны иметь единственное определение в программе.

Синтаксис без аргументов: `int main() {<body>}`

Синтаксис с аргументами: `int main(int argc, char* argv[]) {<body>}`

- `argc` - количество аргументов;
- `argv` - массив строкового представления аргументов.
- вместо `char* argv[]` можно использовать `char** argv`.

Пример:

```
int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; ++i) {
        std::cout << "arg " << i << " : " << argv[i] << std::endl;
    }
    return 0;
}
```

Далее необходимо скомпилировать программу и запустить с параметрами:

```
g++ main.cpp && ./a.out some_param=1 -some_flag
```

Дополнительные материалы

Интерактив:

- Хэндбук Яндекса по C++ - для старта
- Интерактивный учебник по C++ - для старта

Видео:

- YouTube канал SimpleCode - для старта
- YouTube канал Ильи Мещерина - для продвинутых
- YouTube канал Константина Владимирова - для мощных

Для ежедневной работы мозга:

- LeetCode
- CodinGame
- CodeForces

Полезные инструменты

- godbolt - онлайн ресурс для компилирования
- quick-bench - онлайн ресурс для сравнения быстродействия
- cppinsights - онлайн ресурс, убирающий синтаксический сахар
- replit - онлайн ресурс, чтобы поделиться кодом

IDE

- Clion
- VS Code
- Visual Studio
- Qt Creator
- Eclipse for C/C++ extensions