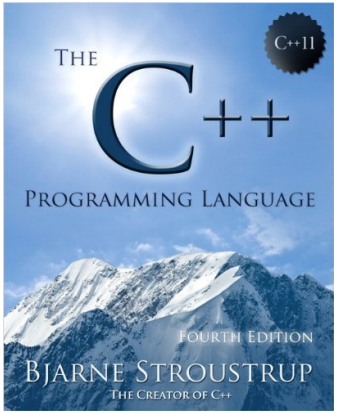
Введение в C++

Литература

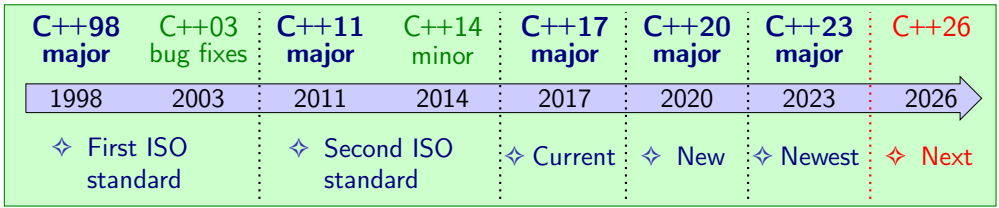
Bjarne Stroustrupe
The C++ Programming Language, **4th Edition**

Полезные ссылки

- C++ reference *: предпочтительно на английском, русская версия неполна*
- A Tour of C++, by Bjarne Stroustrup
- What is the best book to learn C++ from?



Хронология развития C++



Стандарты ISO: International Organization for Standardization

- **C++98** с дополнениями от C++03: «старый, стабильный»
- **C++11** & C++14: широко используемый стандарт
- **C++17**: стандарт поддерживаемый большинством компиляторов
- **C++20**: «новый» стандарт
- **C++23** `(aka c++2b)` «новейший» стандарт (декабрь 2023)
- **C++26** `(aka c++2c)` новые идеи которые только еще тестируют

Поддержка компиляторами

- **GCC (g++):**
  - C++11/14 – полностью начиная с 5.0 (`-std=c++11` или `-std=c++14`)
    в версиях с 6.1 по 10 C++14 стандарт по умолчанию
  - C++17 – полностью с 7.0 (`-std=c++17`),
    с версии 11 C++17 стандарт по умолчанию
  - C++20 – почти полная поддержка в версии 12 (`-std=c++20`)
- **clang (clang++):**
  - C++11/14 – полностью начиная с версии 3.4,
    с clang-6 по clang-15: C++14 стандарт по умолчанию
  - C++17 – полностью с версии 5 (`-std=c++17`),
    начиная с clang-16 C++17 стандарт по умолчанию
  - C++20 – частично, начиная с clang-10 (`-std=c++20`)
- **MS Visual Studio:**
  - C++11/14 – начиная с VS 2015
  - C++17 – начиная с VS 2017 15.8
  - C++20 – полностью (опция `std:c++latest` в VS 2019 16.10)

Стандартная библиотека C в C++

Правила использования в C++

- Имя заголовочного файла такое же как в C, но нет расширения `.h` и добавляется впереди буква `c`: `math.h` $\rightarrow$ `c + math.h` $\rightarrow$ `cmath`
- Все переменные и функции стандартной библиотеки находятся в пространстве имен `std`

Прагматичный подход в C++11

- Рекомендуется: используйте `<cxxx>`, что бы имена гарантированно находились в `std` пространстве
  - ☞ глобальная декларация не гарантирована
- Используйте `<xxx.h>`, что бы имена гарантированно находились в глобальном пространстве имен
  - ☞ декларация в `std` не гарантирована

Заголовочные файлы C++ включают перегрузку функций C:

- для удобства работы с аргументами различных типов:
  `expf() or exp() or expl() -> exp()`
- некоторые функции заменяются на две в C++ для корректности работы с константными указателями

```
char* strchr(const char* s, int c); // only C
char* strchr(char* s, int c);              // C++ #1
const char* strchr(const char* s, int c); // C++ #2
```

☞ Некоторые функции «доопределены» в C++, например `pow(x,n)`

Пространство имен

Концепция пространств имен (Namespaces)

☞ Дает способ для устранения конфликтов имен в больших проектах

Пример: функция для ведения журнала записей (log-file)

```
namespace my_funcs {
  void log(double voltage) { ... };
};
...
my_funcs::log(220.); // записать 220. в журнал; как записать ln(220)?
```

✓ Использовать `std::log()` из пространства имен стандартной библиотеки

```
my_funcs::log( std::log(220.) ); // записать log(220) в журнал
```

Двойное двоеточие :: (the scope resolution operator)

:: – оператор разрешения области видимости

```
std::log(220); // функция log() из пространства std
```

☞ Директива using разрешает использовать короткие имена

1. `using name_space::name` – для одного имени

```
using std::endl;
std::cout << "bla-bla-bla" << endl;
```

2. `using namespace name_space` – все имена из указанного `name_space`

```
using namespace std;
cout << "bla-bla-bla" << endl;
```

☞ «отменить» использование пространства имен невозможно

☞ не путайте с использованием `using` для задания псевдонимов типов

```
using VecI = std::vector<int>;
```

Безымянное пространство имен

- Глобальные переменные находятся в «безымянном пространстве имен»

an unnamed namespace

```
int n = 1;   // a global variable
int main() {
  int n = 2; // a local variable
  cout << "global variable: " << ::n << endl; // global variable: 1
  cout << "local variable: " << n << endl;    // local variable: 2
}
```

Псевдонимы для пространства имен

- возможно задание альтернативного имени для длинного, возможно вложенного пространства имен

```
namespace fs = std::filesystem; // fs is shortcut for std::filesystem
```

Функции в C++

Перегрузка функций

☞ имена функций могут совпадать если каждая функция имеет уникальную сигнатуру вызова

```
int Max(int x, int y);           // (1)
double Max(double x, double y); // (2)
double Max(double arr[]);        // (3)
int Max(double arr[]); // ERROR: = (3)
```

```
int a = Max(3,4);   // call (1)
int b = Max(3.,4.); // call (2)
double ar[] {3.,4.};
double x = Max(ar); // call (3)
```

Задание аргументов функции по умолчанию

☞ Все такие аргументы должны быть справа от обычных аргументов

```
void foo(int x, int y = 10, double z = 2.5);
foo(1,2,3); // обычный вызов: x=1,y=2,z=3
foo(1,2);   // будет вызвана foo(1,2,2.5)
foo(1);     // будет вызвана foo(1,10,2.5)
foo();      // ERROR: x не имеет значения по умолчанию
```

Динамическая память в C++: new и delete

- **new** – оператор выделения динамической памяти
  - ☞ оператор new вызывает конструктор объекта
- **delete** – оператор возврата памяти выделенной с помощью new
  - ☞ оператор delete вызывает деструктор объекта

```
int* pa = new int;                  // allocate one int;
UserClass* pc = new UserClass(1); // allocate UserClass;
*pa = 1;                            // use pa and pc
pc->function(5);
delete pa;                          // destroy pa (int)
delete pc;                          // destroy UserClass
pa = pc = 0;                        // good practice
```

☞ new возвращает указатель имеющий тип

Выделение памяти для массивов, операторы `new[]` и `delete[]`

```
int* pa = new int[10];              // allocate 10 int's
UserClass* pc = new UserClass[5]; // allocate 5 UserClass's
// Note: pa (pc) is the pointer to first element of array
for(int i = 0; i < 5; i++ ) {
   pa[i] = i*i;
   pc[i]->function(i);
}
delete[] pa;                        // destroy arrays pa,
delete[] pc;                        //                 pc
pa = pc = 0;                        // good practise
```

Обратите внимание

- ☞ Удаление одного объекта – `delete`, массива – `delete[]`
- ☞ Для создания массива объектов класса необходим конструктор без аргументов – «конструктор по умолчанию»: `UserClass()`

Сравнение new и delete с malloc(), calloc() и free()

- `new, delete, new[], delete[]` – операторы C++
  - `malloc(), calloc() и free()` – функции C-stdlib
- `new` возвращает тип «указатель на класс»
  - `malloc() и calloc()` возвращают тип `void*`

☞ `malloc() и calloc()` не умеют вызывать конструкторы, а `free()` не умеет вызывать деструктор

☞ `delete, delete[]` – вызывают деструктор автоматически, явно вызывать деструктор не надо

- если оператор `new` не может выделить память, то возбуждается исключение типа `std::bad_alloc`

Библиотеки ввода-вывода <iostream> и <cstdio>

```
#include <iostream>  // C++ input/output lib header
#include <cstdio>    // C compatibility header for std::printf

int main() {
   std::cout << "Enter the number: "; // output
   int num = 0;
   std::cin >> num;                   // input 10
   size_t sum = 0;
   for(int i = 1; i < num; i++) {sum += i*i*i;}
   std::cout << "The sum of cubes of natural numbers from 1 to "
      << num << " is " << sum << std::endl;
   // The sum of cubes of natural numbers from 1 to 10 is 2025

   std::printf("Σn³ = %zu\n",sum); // Σn³ = 2025
}
```

Потоки в C++

- **cout** – стандартный поток вывода, буферизованный (`stdout` в C)
- **cin** – стандартный поток ввода, буферизованный (`stdin` в C)
- **cerr** – поток сообщений об ошибках, небуферизованный (`stderr` в C)
- **clog** – буферизованный вариант `cerr`

Обратите внимание

- ☞ Переменные `cout, cin, cerr, endl ...` определены в пространстве имен стандартной библиотеки `std`
- ☞ Функции из `iostream` и `cstdio` можно использовать одновременно, однако не стоит забывать что буферизация в них может быть выполнена раздельно

Универсальная инициализация (C++11)

«Инициализация списком» в фигурных скобках {}

```
double a = 1.2;              // обычная инициализация
double a {1.2};              // new C++11
double a = {1.2};            // в C++11 это новая инициализация
char c[] {"abc"};            // инициализация массива
vector<int> v {1,2,3,4,5}; // вектора
```

Универсальная инициализация более строгая

```
int a = 1.2;   // a = 1 (warning in the best case)
int a {1.2};   // ERROR: 'double' cannot be narrowed to 'int'
char ch {332}; // ERROR: narrowing conversion
double d {a};  // warning: narrowing conversion 'int' to 'double'
```

Тип переменной при инициализации (C++11)

Placeholder type specifier `auto`

- тип переменной определяется по типу правой части:

```
auto a = 1.2;      // a - double
auto b = 1;        // b - int
auto x = fun(b);   // x - тип который возвращает fun(int)
```

- осторожно с универсальной инициализацией: (since C++17)

```
auto d   {1};   // int
auto e = {1};   // std::initializer_list<int>
auto f   {1,2}; // ERROR: not a single element
auto g = {1,2}; // OK: std::initializer_list<int>
```

- после auto может быть несколько переменных:

```
auto i = 1, *p = &i, &r = i; // int, int* and int&
const auto j = 2L, *pj = &j; // const long and const long*
```

```
auto int x = 10; // valid in C and C++98, error as of C++11
```

auto in structured binding declaration (C++17)

☞ Привязка типа переменных при инициализации «структурой»

```
int ar[2] {1, 2};     // array
auto& [xr,yr] = ar;  // xr refers to ar[0], yr refers to ar[1]
cout << "xr=" << xr << " yr=" << yr << '\n'; // xr=1 yr=2

struct C {int x=1; double y=2.1;}; // class declaration
auto [xs,ys] = C();                // binding to data members
cout << "xs=" << xs << " ys=" << ys << '\n'; // xs=1 ys=2.1

auto tuple = std::make_tuple(1, 2.2, 'C');
auto [i,d,c] = tuple; // type declaration and unpacking
cout << "i=" << i << " d=" << d << " c=" << c << '\n'; // i=1 d=2.2 c=C
```

Порядок вычисления в C++17

Замечательный проблемный пример

```
std::string str = "I heard it even works if you don't believe";
//we want to have:"it sometimes works if I believe"

str.replace(0,8,"")                              // "I heard "
   .replace(str.find("even"),4,"sometimes") // "even"->"sometimes"
   .replace(str.find("you don't"),9,"I");   // "you don't"->"I"
std::cout << str << '\n';
// result: -std=c++17: it sometimes works if I believe
// gcc-4.8 -std=c++11: it even worsometimesf youllieve
```

☞ Порядок вычисления аргументов функции в C++ не определен

```
int a() {return std::puts("a");}
int b() {return std::puts("b");}
int c() {return std::puts("c");}
void z(int,int,int){};
```

```
int main() {
  z(a(),b(),c());
  // all combinations are allowed
} // gcc: cba clang: abc
```

Более строгие правила вычисления в C++17

- В этих выражениях гарантируется, что E1 вычисляется до E2

```
E1(E2)          E2=E1
E1[E2]          E2+=E1
E1.E2           E2-=E1
E1->E2          E2*=E1
E1<<E2          E2/=E1
E1>>E2
```

- Если в выражении fun(a(x),b(y),c(z)) вычисляется y то b(y) вычисляется полностью, а затем x,a(x) или z,c(z)

```
int i = 0;
std::cout << ++i << ' ' << --i << '\n'; // 1 0 garanted in C++17
i = i++ + 2;                           // well-defined in C++17
i = i++ + i;                           // still undefined
```

Дополнительные слайды

Программа “Hello world!”

hello.cpp

```
// Hello World in C++
#include <iostream>
using namespace std;
int main() {
  cout<<"Hello, world!"<<endl;
}
```

hello.c

```
/* Hello World in C */
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
}
```

Обратите внимание!

- ☞ В именах стандартных заголовочных файлов C++ нет `.h`
  - «Волшебная» строка: `using namespace std`
  - В C++ как и в C99, в конце `main()` неявно определен `return 0;`

Компиляция

- Расширение имени файла для C++:
  .cpp `.C` `.cc` `.cxx` `.c++` `...`
- Вызов компилятора:
  `COMMAND_PROMPT> clang++ hello.cpp`
  (или для gcc `COMMAND_PROMPT> g++ hello.cpp`)
  (или «определяемое системой» `COMMAND_PROMPT> c++ hello.cpp`)
- Выполнение:
  `COMMAND_PROMPT> ./a.out`
  `Hello, world!`

Проверка (assertion) при компиляции (C++11)

`static_assert(bool_constexpr, message)`

☞ Проверка логического выражения во время компиляции и остановка компиляции если выражение ложно

`bool_constexpr` — константное выражение

`message` — текст выводимый при остановке, в C++17 необязателен

Проверка, что -3/2 == -1 и продолжение если это истина

```
static_assert(-3/2==-1, "negative values rounds away from zero");
static_assert(int(-3./2)==-1,"negative values rounds away from zero");
```

В случае неудачи – остановка компиляции

```
static_assert(sizeof(void*)<=sizeof(int), "can not store void* in int");
error: static assertion failed: can not store void* in int
```