

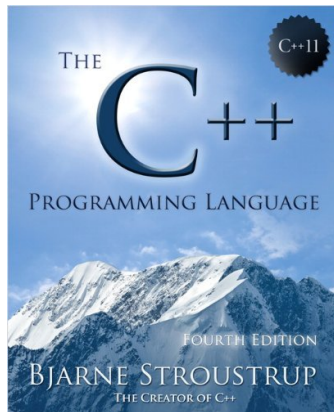
Введение в C++

Литература

Bjarne Stroustrup
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 C++

C++98 major	C++03 bug fixes	C++11 major	C++14 minor	C++17 major	C++20 major	C++23 major	C++26
1998	2003	2011	2014	2017	2020	2023	2026
✧ First ISO standard		✧ Second ISO standard		✧ Current	✧ New	✧ Newest	✧ Next

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)

Основные особенности C++

C++ язык программирования общего назначения

- поддерживающий абстракцию данных
- поддерживающий объектно-ориентированное программирование (ООП)
- поддерживающий обобщенное программирование (Generic)
- «улучшает» язык C

Направление развития

- Обеспечить совместимость с C99 и с предыдущими версиями C++
- Сделать C++ проще для изучения
- Сделать C++ более подходящим для системного программирования и разработки библиотек

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!`

От C к C++: улучшения

Знакомы по C99!

- Новый стиль комментариев: `//`
 - Целые типы из C99: `int8_t ... int64_t ...` (C++11)
 - Встраиваемые (`inline`) функции:
`inline double SQ(double x) {return x*x;}`
 - Переменные рекомендуется создавать по мере необходимости, например внутри цикла `for(int i = 0, ...)`
- ☞ В C++ считается, что использование препроцессорных макросов – зло!

Старый и новый стиль комментариев: что будет напечатано?

```
printf("%d", -1 /**/ 2  
+ 1);
```

C++: новые возможности

Перегрузка функций

- имена функций могут совпадать если каждая функция имеет уникальную **сигнатуру вызова**
- ☞ одинаковые имена **подразумевают** выполнение концептуально одинаковых задач

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

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


Задание аргументов функции по умолчанию

- Для аргументов функции можно задавать значения 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
```

Аргументы без имени

- Если аргумент не используется, имя не обязательно

```
double foo2(int, int y, int z) { // первый аргумент не используется  
    return sqrt(y*y + z*z);  
}
```

C++: библиотека ввода-вывода `iostream`

```
#include <iostream>    // header for C++ input/output lib
#include <cmath>        // not math.h !
using namespace std;
int main() {
    cout << " Enter the number: ";    // output
    int num = 0;
    cin >> num;                       // input
    double sum = 0.;
    for(int i = 1; i < num; i++) sum += sqrt(i);
    cout << " The sum of square roots of numbers from 0 to "
         << num << " is " << sum << endl;
}
```

Enter the number: 10

The sum of square roots of numbers from 0 to 10 is 19.306

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`, `clog` определены в пространстве имен стандартной библиотеки `std`:
- 1 полные имена: `std::cout`, `std::cin`, `std::endl` ...
 - 2 «магическая» директива `using namespace std`; позволяет использовать короткие имена

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

❶ using name_space::name : для **одного** имени

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

❷ using namespace name_space : **все имена** из указанного name_space

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

👉 «ОТМЕНИТЬ» использование пространства имен невозможно

An unnamed namespace

👉 Глобальные переменные находятся в «безымянном пространстве имен»

Пример

```
#include <iostream>
using namespace std;
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
}
```

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` → `c + math/`~~`h`~~ → `cmath`
- Все переменные и функции стандартной библиотеки находятся в `std::`

Новый, прагматичный подход в C++11

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

`const char* strchr(const char* s, int c);`

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

Проблема с `abs()`:

исправлено в C++11

- 🔗 в `<stdlib.h>` определены `abs()`, `labs()` и `llabs()`;

в `<math.h>` соответственно `fabs()`, `fabsf()`, `fabsl()`;

в C++ `abs()` должна работать с любым типом, но это не работает если подключен только один заголовочный файл

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 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 деструкторы автоматически, явно вызывать деструктор не надо

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

Указатель без объекта:



- C: `*ptr = NULL;` // #define NULL 0 (in stdlib.h)
- C++: `*ptr = 0;` // instead of NULL
- C++11: `*ptr = nullptr;` // `std::nullptr_t` type

Зачем `nullptr` нужен?

```
void func(int n)
void func(char* s) // two overloaded func() in C++
...
func(0);           // guess which function gets called? ... func(int)
func(nullptr);     // no doubt: C++11
func((char*) 0);   // no doubt: C++98
```


C++: новые типы данных

Логический тип `bool` — переменные могут иметь два значения

 `false` } в арифметических выражениях `false` \rightarrow 0 и `true` \rightarrow 1
 `true`

Типы `struct`, `enum` и `union`


```
struct element { char name[50]; int number; double A; }  
enum ECOLOR { red, green, blue };
```

 Создание новых переменных (объектов):

```
element H;           // C++  
ECOLOR linecolor;    // C++
```

```
||  
struct element H;     // C  
enum ECOLOR linecolor; // C
```

Перечисления со строгой типизацией `enum struct` или `enum class`

- Общий вид: `enum struct name : type { enumerator1 = constexpr, enumerator2 = constexpr, ... }`
- `enum struct` \equiv `enum class`  и не имеют отношения к классам!
- `constexpr` – константное выражение

Имеет множество преимуществ перед обычным `enum`

- 1 Имеет собственное пространство имен – имя `enum`-класса:

```
enum Animals { Cat=0, Dog=1, Chicken=2 };  
enum Birds { Duck=0, Chicken=1 }; // ERROR! redeclaration of Chicken
```

```
enum class Fruits { Apple=1, Orange=2 }; // no error: Fruits::Orange  
enum class Colors { Red=1, Orange=2 }; //      && Colors::Orange
```

- ② Запрещено неявное преобразование к `int`:

```
enum Animals { Cat, Dog, Chicken };  
bool b = Cat && Dog;    // what?
```

```
enum class Fruits { Apple, Orange };  
int e = Fruits::Orange; // ERROR! cannot convert  
int e = int(Fruits::Orange); // OK!
```

- ③ Можно указать целый тип лежащий в основе перечисления:

```
enum MyBits { B1 = 0x01, B2 = 0x10, Bbig = 0xFFFFFFFF };  
// no guaranty for Bbig: implementation defined!
```

```
enum class MyBits : unsigned long long  
    {B1 = 0x01ULL, B2 = 0x10ULL, Bbig = 0xFFFFFFFFFULL }; // OK!
```

Days 1 - 10

Teach yourself variables, constants, arrays, strings, expressions, statements, functions,...



Days 11 - 21

Teach yourself program flow, pointers, references, classes, objects, inheritance, polymorphism,



Days 22 - 697

Do a lot of recreational programming. Have fun hacking but remember to learn from your mistakes.



Days 698 - 3648

Interact with other programmers. Work on programming projects together. Learn from them.



Days 3649 - 7781

Teach yourself advanced theoretical physics and formulate a consistent theory of quantum gravity.



Days 7782 - 14611

Teach yourself biochemistry, molecular biology, genetics,...



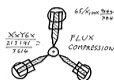
Day 14611

Use knowledge of biology to make an age-reversing potion.



Day 14611

Use knowledge of physics to build flux capacitor and go back in time to day 21.



Day 21

Replace younger self.



As far as I know, this is the easiest way to

"Teach Yourself C++ in 21 Days".