Лекция 1 – введение в язык C++

Воронин Андрей Андреевич

Кафедра прикладной математики и информатики

18 сентября 2019 г.

Язык C

- Язык программирования C разработан в начале 1973 года в компании Bell Labs Кеном Томпсоном и Деннисом Ритчи.
- Язык C был создан для использования в операционной системе UNIX.
- В связи с успехом UNIX язык C получил широкое распространение.
- На данный момент C является одним из самых распространенных языков программирования (доступен на большинстве платформ).
- C – основной язык низкоуровневой разработки.
- Язык программирования C++ создан на основе языка C.

Особенности языка C

- **Эффективность.**
  Язык C позволяет писать программы, которые напрямую работают с железом.
- **Стандартизированность.**
  Спецификация языка C является международным стандартом.
- **Относительная простота.**
  Стандарт языка C занимает 520 страниц (Java 772, C++ 1605).

Создание C++

- Разрабатывается с начала 1980-х годов.
- Создатель – сотрудник Bell Labs Бьёрн Страуструп.
- Изначально это было расширение языка C для поддержки работы с классами и объектами.
- Это позволило проектировать программы на более высоком уровне абстракции.
- Ранние версии языка назывались ”C with classes”.
- Первый компилятор cfront, перерабатывал исходный код ”C with classes” в исходный код на C.

Развитие C++

- К 1983 году в язык были добавлены множество новых возможностей (виртуальные функции, перегрузка функций и операторов, ссылки, константы, ...)
- Получившийся язык перестал быть просто дополненной версией классического C и был переименован из ”C with classes” в C++.
- Имя языка, получившегося в итоге, происходит от оператора унарного постфиксного инкремента C ’++’ (увеличение значения переменной на единицу).
- Язык также не был назван D, поскольку ”является расширением C и не пытается устранять проблемы путем удаления элементов C”.
- Язык начинает активно развиваться. Появляются новые компиляторы и среды разработки.

Стандартизация C++

- Лишь в 1998 году был ратифицирован международный стандарт языка C++: ISO/IEC 14882:1998 ”Standard for the C++ Programming Language”.
- В 2003 году был опубликован стандарт языка ISO/IEC 14882:2003, где были исправлены выявленные ошибки и недочеты предыдущей версии стандарта.
- В 2005 году был выпущен Library Technical Report 1 (TR1).
- С 2005 года началась работа над новой версией стандарта, которая получила кодовое название C++0x.
- В конце концов в 2011 году стандарт был принят и получил название C++11 ISO/IEC 14882:2011.
- C++14 ISO/IEC 14882:2014 небольшое расширение и исправление ошибок предыдущего стандарта.
- Стандарт C++17 ISO/IEC 14882:2017 принес изменения направленные на большую безопасность языка.
- C++20 находится в разработке и обещает серьёзный набор изменений включающих изменение процесса сборки.

Совместимость C и C++

- Один из принципов разработки стандарта C++ – это сохранение совместимости с C.
- Синтаксис C++ унаследован от языка C.
- C++ в строгом смысле не является надмножеством C.
- Можно писать программы на C так, чтобы они успешно компилировались на C++.
- C и C++ сильно отличаются как по сложности, так и по принятым архитектурным решениям, которые используются в обоих языках.

Сложность C++

C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.
(В языке C легко прострелить себе ногу. В C++ это сложнее, но если вы сделаете это, то отстрелите всю ногу целиком.)

Bjarne Stroustrup

Every extension proposal should be required to be accompanied by a kidney. People would submit only serious proposals, and nobody would submit more than two.
(Нужно, чтобы к каждому предложению о расширении языка обязательно прилагалась почка. Тогда люди присылали бы только очень важные предложения, и никто не прислал бы более двух.)

Jim Waldo

Сложность C++

- Описание стандарта занимает 1605 страниц текста.
- Нет никакой возможности рассказать ”весь C++” в рамках одного, путь даже очень большого курса.
- В C++ программисту позволено делать очень многое, и это влечет за собой большую ответственность.
- На плечи программиста ложится много дополнительной работы:
  - проверка корректности данных,
  - управление памятью,
  - обработка низкоуровневых ошибок.

Мультипарадигменность

На языке C++ можно писать программы в рамках нескольких парадигм программирования:

- процедурное программирование (код ”в стиле C”);
- объектно-ориентированное программирование (классы, наследование, виртуальные функции, ...);
- обобщенное программирование (шаблоны функций и классов);
- функциональное программирование (функторы, безымянные функции, замыкания);
- генеративное программирование (метапрограммирование на шаблонах).

Эффективность

Одна из фундаментальных идей языков C и C++ – отсутствие неявных накладных расходов, которые присутствуют в других более высокоуровневых языках программирования.

- Программист сам выбирает уровень абстракции, на котором писать каждую отдельную часть программы.
- Можно реализовывать критические по производительности участки программы максимально эффективно.
- Эффективность делает C++ основным языком для разработки приложений с компьютерной графикой (к примеру, игры).

Низкоуровневость

Язык C++, как и C, позволяет работать напрямую с ресурсами компьютера.

- Позволяет писать низкоуровневые системные приложения (например драйверы операционной системы).
- Неаккуратное обращение с системными ресурсами может привести к падению программы.

В C++ отсутствует автоматическое управление памятью.

- Позволяет программисту получить полный контроль над программой.
- Необходимость заботиться об освобождении памяти.

Компилируемость

C++ является компилируемым языком программирования. Для того, чтобы запустить программу на C++, ее нужно сначала *скомпилировать*. Компиляция – преобразование текста программы на языке программирования в машинный код.

- Нет накладных расходов при исполнении программы.
- При компиляции можно отловить некоторые ошибки.
- Требуется компилировать для каждой платформы отдельно.

Статическая типизация

C++ является статически типизированным языком.

1. Каждая сущность в программе (переменная, функция и пр.) имеет свой тип,
2. и этот тип определяется на момент компиляции.

Это нужно для того чтобы

1. вычислить размер памяти, который будет занимать каждая переменная в программе,
2. определить, какая функция будет вызываться в каждом конкретном месте.

Всё это определяется на момент компиляции и ”зашивается” в скомпилированную программу. В машинном коде никаких типов уже нет, там идет работа с последовательностями байтов.

Выберите все верные утверждения из списка.

- ☐ C++ ориентирован на написание эффективных приложений.
- ☐ C++ поддерживает объектно-ориентированное программирование.
- ☐ C++ не поддерживает процедурное программирование.
- ☐ C++ поддерживает процедурное программирование.
- ☐ C++ интерпретируемый язык программирования
- ☐ C++ ориентирован на безопасность работы с памятью.

Выберите все верные утверждения из списка.

- ☒ C++ ориентирован на написание эффективных приложений.
- ☒ C++ поддерживает объектно-ориентированное программирование.
- ☐ C++ не поддерживает процедурное программирование.
- ☒ C++ поддерживает процедурное программирование.
- ☐ C++ интерпретируемый язык программирования
- ☐ C++ ориентирован на безопасность работы с памятью.

Что такое компиляция?

Проектирование → Архитектура программы

Программирование → Код на C++

Компиляция → Машинный код

Исполнение →

Что такое компиляция?

Проектирование

Архитектура программы

Программирование

Код на Java

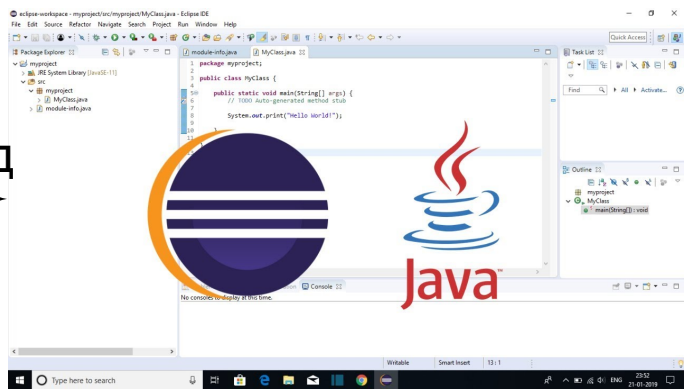
Компиляция

Байт код

Исполнение

Java Virtual Machine (JVM)

Трансляция команд



Что такое компиляция?

Плюсы и минусы компилируемости в машинный код

Плюсы

- эффективность: программа компилируется и оптимизируется для конкретного процессора
- нет необходимости устанавливать сторонние приложения (такие как интерпретатор или виртуальная машина).

Минусы

- нужно компилировать для каждой платформы
- сложность внесения изменений в программу – нужно перекомпилировать заново.

Важно: компиляция – преобразование одностороннее, нельзя восстановить исходный код.

Выберите все верные утверждения из списка.

- ☐ Код программы, написанный на интерпретируемом языке, можно без предварительной компиляции запустить на любой платформе, где есть интерпретатор этого языка.
- ☐ Код программы, написанный на языке, который компилируется в байт код виртуальной машины, достаточно скомпилировать однажды, чтобы программу можно было запускать на любой платформе, где есть соответствующая виртуальная машина.
- ☐ Для запуска программы, код которой был написан на компилируемом языке, на компьютере должен быть установлен компилятор этого языка.
- ☐ Код программы, написанный на языке, который компилируется в машинный код, достаточно скомпилировать однажды, и потом программу можно будет запустить на любой платформе.
- ☐ Для запуска программы, код которой был написан на интерпретируемом языке, на компьютере должен быть установлен интерпретатор этого языка.
- ☐ Скомпилировать программу на C++ для некоторой архитектуры X можно только на компьютере с архитектурой X.

Выберите все верные утверждения из списка.

- ☒ Код программы, написанный на интерпретируемом языке, можно без предварительной компиляции запустить на любой платформе, где есть интерпретатор этого языка.
- ☒ Код программы, написанный на языке, который компилируется в байт код виртуальной машины, достаточно скомпилировать однажды, чтобы программу можно было запускать на любой платформе, где есть соответствующая виртуальная машина.
- ☐ Для запуска программы, код которой был написан на компилируемом языке, на компьютере должен быть установлен компилятор этого языка.
- ☐ Код программы, написанный на языке, который компилируется в машинный код, достаточно скомпилировать однажды, и потом программу можно будет запустить на любой платформе.
- ☒ Для запуска программы, код которой был написан на интерпретируемом языке, на компьютере должен быть установлен интерпретатор этого языка.
- ☐ Скомпилировать программу на C++ для некоторой архитектуры X можно только на компьютере с архитектурой X.

Разбиение программы на файлы

Зачем разбивать программу на файлы?

- С небольшими файлами удобнее работать.
- Разбиение на файлы структурирует код.
- Позволяет нескольким программистам разрабатывать приложение одновременно.
- Ускорение повторной компиляции при небольших изменениях в отдельных частях программы.

Файлы с кодом на C++ бывают двух типов:

1. файлы с исходным кодом (расширение `.cpp`, `.cxx`, `.cc` иногда `.C`)
2. заголовочные файлы (расширение `.hpp`, `hxx`, `hh` или `.h`)

Заголовочные файлы

Определение функции

Файл `foo.cpp`:

```
// определение (defenition) функции foo
void foo()
{
    bar();
}
```

Файл `bar.cpp`:

```
// определение (defenition) функции bar
void bar(){ }
```

Компиляция этих файлов вызовет ошибку.

Заголовочные файлы

Объявление и определение

Файл `foo.cpp`:

```
// объявление (declaration) функции bar
void bar();

// определение (defenition) функции foo
void foo()
{
    bar();
}
```

Файл `bar.cpp`:

```
// определение (defenition) функции bar
void bar(){ }
```

Заголовочные файлы

Неопределенное поведение

Файл `foo.cpp`:

```
void bar();

void foo()
{
    bar();
}
```

Файл `bar.cpp`:

```
// определение (defenition) функции bar
int bar(){ return 1; }
```

Данный код некорректен – объявление отличается от определения. (Неопределенное поведение.)

Заголовочные файлы

Подключение заголовочных файлов

Файл `foo.cpp`:

```
#include "bar.hpp"

void foo()
{
    bar();
}
```

Файл `bar.cpp`:

```
int bar(){ return 1; }
```

Файл `bar.hpp`:

```
int bar();
```

Заголовочные файлы

Двойное включение

Может случиться двойное включение заголовочного файла. Файл `foo.cpp`:

```
#include "foo.hpp"
#include "bar.hpp"

void foo()
{
    bar();
}
```

Файл `foo.hpp`:

```
#include "bar.hpp"

int foo();
```

Заголовочные файлы

Защита от двойного включения

Предыдущую ситуацию можно исправить двумя способами:

- (наиболее переносимо) Файл `bar.hpp`:

```
#ifndef BAR_HPP
#define BAR_HPP

int bar();

#endif //BAR_HPP
```

- (наиболее просто) Файл `bar.hpp`:

```
#pragma once

int bar();
```

Объявление и определение

Объявление (declaration) — вводит имя, возможно, не определяя деталей. Например, ниже перечислены объявления:

- `int a;`
- `void foo();`
- `void bar() { foo(); }`

Определение (definition) — это объявление, дополнительно определяющее детали, необходимые компилятору. Из перечисленных выше объявлений, определениями являются только два:

- `int a;`
- `void bar() { foo(); }`

Для определения переменной достаточно указать ее тип, а для определения функций, кроме имени, типов параметров и возвращаемого значения, нужно указать еще тело функции. Проще говоря, определение содержит всю информацию, необходимую компилятору, чтобы выделить память для хранения объекта.
В C++ есть также возможность объявить переменную, не определяя ее:
`extern int a;`

Объявление и определение

- `extern int a;` – объявление переменной типа `int`,
- `int a;` – объявление и **определение** переменной типа `int`,
- `void foo();` – объявление функции с именем `foo`
- `void bar(){ foo(); }` – объявление и **определение** функции с именем `bar`

Интересно отметить, что файлы стандартной библиотеки C++ не используют расширение вовсе, например:

- iostream,
- algorithm,
- vector.

Разделение на файлы с исходным кодом и заголовочные файлы чисто условное, нет правил, запрещающих использовать .cpp файл как заголовочный, однако мы не рекомендуем так делать — использование общепринятых правил именования файлов упростит жизнь вам и вашим коллегам.
Не стоит помещать *определения* в заголовочные файлы без явной необходимости. В C++ есть способы, позволяющие поместить определение в заголовочный файл, не вызвав при этом ошибки компоновщика, но, как правило, это приводит к увеличению объектного файла и программы в целом

Выберите из списка объявления, которые **не** стоит помещать в заголовочные файлы.

- ☐ `void foo() { std::cout << "Hello, World!\n "; }`
- ☐ `void foo();`
- ☐ `int a;`
- ☐ `extern int a;`
- ☐ `void bar() { foo(); }`

Выберите из списка объявления, которые **не** стоит помещать в заголовочные файлы.

- ☒ `void foo() { std::cout << "Hello, World!\n "; }`
- ☐ `void foo();`
- ☒ `int a;`
- ☐ `extern int a;`
- ☒ `void bar() { foo(); }`

Этап №1: Препроцессор

- Язык препроцессора – это специальный язык программирования, встроенный в C++.
- Препроцессор работает с кодом на C++ как с текстом.
- Команды языка препроцессора называют директивами, все директивы начинаются со знака `#`.
- Директива `#include` позволяет подключать заголовочные файлы к файлам кода.
  1. `#include <foo.h>` – библиотечный заголовочный файл,
  2. `#include "bar.h"` – локальный заголовочный файл.
- Препроцессор заменяет директиву `#include "bar.h"` на содержимое файла `bar.h`.

Этап №2: Компиляция

- На вход компилятору поступает код на C++ после обработки препроцессором.
- Каждый файл с кодом компилируется отдельно и независимо от других файлов с кодом.
- Компилируются только файлы с кодом (т.е. `*.cpp`).
- Заголовочные файлы сами по себе ни во что не компилируются, только в составе файлов с кодом.
- На выходе компилятора из каждого файла с кодом получается ”объектный файл” – бинарный файл со скомпилированным кодом (с расширением `.o` или `.obj`).

Этап №3: Линковка (компоновка)

- На этом этапе все объектные файлы объединяются в один исполняемый (или библиотечный) файл.
- Пр этом происходит подстановка адресов функций в места их вызова.

```
void foo()
{
    bar();
}
```

```
void bar() { }
```

- По каждому объектному файлу строится таблица всех функций, которые в нем определены.

Этап №3: Линковка (компоновка)

- На этапе компоновки важно, что каждая функция имеет уникальное имя.
- В C++ может быть две функции с одним именем, но разными параметрами.
- Имена функций искажаются (mangle) таким образом, что в их имени кодируются их параметры.
  Например, компилятор GCC превратит имя функции `foo`

```
void foo(int a, double b) {}
```

  в `_Z3fooid`

Этап №3: Линковка (компоновка)

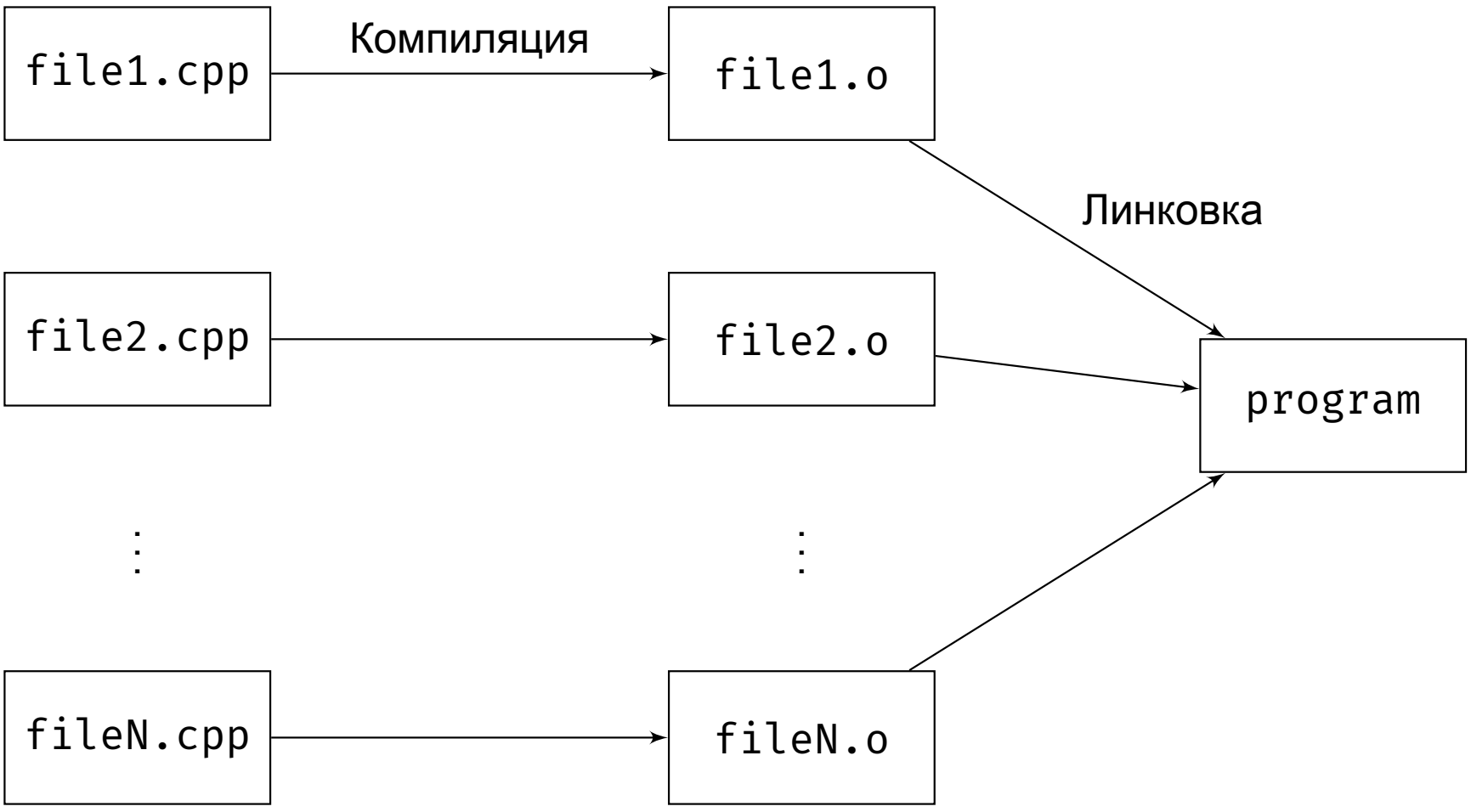
- *Точка входа* – функция, вызываемая при запуске программы. По умолчанию – это функция `main`:

```
int main()
{
    return 0;
}
```

или

```
int main(int argc, char **argv)
{
    return 0;
}
```

Общая схема


file1.cpp
Компиляция
file1.o
Линковка
file2.cpp
file2.o
program
fileN.cpp
fileN.o

Выберите все верные утверждения из списка.

- ☐ Даже для программы состоящей из одной пустой функции `int main() { return 0; }` все равно требуется линковка.
- ☐ Если в коде C++ вы вызываете необъявленную функцию, то это не ошибка, при условии, что функция была где-то определена.
- ☐ Если в коде C++ вы вызываете функцию, которая была объявлена, но не была определена, то это ошибка этапа компиляции.
- ☐ Для программы, состоящей всего из одного файла, не требуется линковка.
- ☐ Если в коде C++ вы вызываете необъявленную функцию, то это ошибка этапа компиляции.
- ☐ Если в коде C++ вы вызываете функцию, которая была объявлена, но не была определена, то это ошибка этапа линковки.

Выберите все верные утверждения из списка.

- ☒ Даже для программы состоящей из одной пустой функции `int main() { return 0; }` все равно требуется линковка.
- ☐ Если в коде C++ вы вызываете необъявленную функцию, то это не ошибка, при условии, что функция была где-то определена.
- ☐ Если в коде C++ вы вызываете функцию, которая была объявлена, но не была определена, то это ошибка этапа компиляции.
- ☐ Для программы, состоящей всего из одного файла, не требуется линковка.
- ☒ Если в коде C++ вы вызываете необъявленную функцию, то это ошибка этапа компиляции.
- ☒ Если в коде C++ вы вызываете функцию, которая была объявлена, но не была определена, то это ошибка этапа линковки.

Типы данных

- Целочисленные:
  1. **`char`**
  2. **`short int`**
  3. **`int`**
  4. **`long int`**
  5. **`long long int`**

  Могут быть беззнаковыми (**`unsigned`** или **`uint8_t`**)
  - $-2^{n-1} \dots (2^{n-1} - 1)$ (n – число бит)
  - $0 \dots (2^n - 1)$ (для **`unsigned`**)
- Числа с плавающей точкой:
  1. **`float`**, 4 байта, 7 значащих цифр.
  2. **`double`**, 8 байт, 15 значащих цифр.
- Логический тип данных **`bool`**.
- Пустой тип **`void`**.

Типы данных

- Целочисленные:
  1. **`int8_t`**
  2. **`int16_t`**
  3. **`int32_t`**
  4. **`int32_t`**
  5. **`int64_t`**

  Могут быть беззнаковыми (**`unsigned`** или **`uint8_t`**)
  - $-2^{n-1} \dots (2^{n-1} - 1)$ (n – число бит)
  - $0 \dots (2^n - 1)$ (для **`unsigned`**)
- Числа с плавающей точкой:
  1. **`float`**, 4 байта, 7 значащих цифр.
  2. **`double`**, 8 байт, 15 значащих цифр.
- Логический тип данных **`bool`**.
- Пустой тип **`void`**.

Литералы

- Целочисленные:
  1. `'a'` – код буквы `'a'`, тип **`char`**.
  2. `42` – все целые по умолчанию типа **`int`**.
  3. `123456789L` – суффикс `'L'` соответствует типу **`long`**.
  4. `1703U` – суффикс `'U'` соответствует типу **`unsigned int`**.
  5. `1703UL` – суффикс `'UL'` соответствует типу **`unsigned long`**.
- Числа с плавающей точкой:
  1. `3.14` – все числа с точкой по умолчанию **`double`**.
  2. `3.1415F` – суффикс `'F'` соответствует типу **`float`**.
  3. `3.0E8` – соответствует $3 \cdot 10^8$
- `true` `false` – значения типа **`bool`**.
- Строки задаются в двойных кавычках: `"Test string"`.

Переменные

- При определении переменной указывается её тип. При определении можно сразу задать начальное значение (инициализация).

```
int i = 10;
short j = 20;
bool b = false;

unsigned long l = 123321;

double x = 13.5, y = 3.1415;
float z;
```

- Нельзя оставлять переменные неинициализированными.
- Нельзя создать переменную пустого типа **void**.

Операции

- Оператор присваивания: **=**.
- Арифметические:
  1. бинарные: **+ - * / %**,
  2. унарные: **++ --**.
- Логические:
  1. бинарные: **&& ||**
  2. унарные: **!**.
- Сравнения: **== != > < >= <=**.
- Приведения типов: **(type)**.
- Сокращенные версии бинарных операторов: **+= -= *= /= %=**.

```
int i = 10;
i = (20 * 3) % 7;

int k = i++;
int l = --i;

bool b = !(k == 1);

b = (a == 0) || (1 / a <
↪   1);

double d = 3.1415;
float f = (int) d;

// d = d * (i + k);
d *= i + k;
```

Целочисленные типы в C++

Все целочисленные типы (кроме `char`) являются знаковыми. Беззнаковые версии типов определяется с ключевым словом `unsigned`, например: `unsigned short int`, `unsigned int` или `unsigned long int`. Для симметрии в языке предусмотрено явное указание того, что тип является знаковым — ключевое слово `signed` (используется редко). Более того, C++ допускает использование следующих сокращений:

- `unsigned` вместо `unsigned int`,
- `short` вместо `short int`,
- `long` вместо `long int`.

Операции инкремента и декремента:

Операции инкремента и декремента

```
int a = 10; // a = 10
int b = ++a; // префиксный инкремент возвращает новое
↪  значение => b = 11 и a = 11
int c = a++; // постфиксный инкремент возвращает
↪  старое значение => c = 11 и a = 12
```

Преобразования типов в операторах

Выражениям так же как и значениям в C++ приписывается некоторые типы. Например, если `a` и `b` — это переменные типа `int`, то выражения `(a + b)`, `(a - b)`, `(a * b)` и `(a / b)` тоже будут иметь тип `int`. Важно всегда понимать, какой тип у выражения, которое вы написали в программе. Давайте проиллюстрируем это на следующем примере:

```
int a = 20;
int b = 50;
double d = a / b; // d = 0, оба аргумента
 ↪  целочисленные, а значит деление целочисленное
```

Для того чтобы исправить эту ситуацию хотя бы один из аргументов оператора деления должен иметь типа double. Этого можно добиться при помощи уже известного нам оператора приведения типов:

```
double d = (double)a / b; // d = 0.4
```

Почему это сработало? Дело в том, что операторы для встроенных типов C++ *всегда* работают с *одинаковыми типами* аргументов. Если аргументы имеют разные типы, то происходит преобразование типов (promotion).

Правило преобразования встроенных типов в операторах

Рассмотрим выражение `(a + b)`, где вместо `'+'` может стоять любой другой подходящий оператор.

1. Если один из аргументов имеет числовой тип с плавающей точкой, то второй аргумент приводится к этому типу (например, при сложении `double` и `int` значение типа `int` приводится к `double`).
2. Если оба аргумента имеют числовой тип с плавающей точкой, то выбирается наибольший из этих типов (например, при сложении `double` и `float` значение типа `float` приводится к `double`).
3. Если оба аргумента целочисленные, но их типы меньше `int`, то оба аргумента приводятся к типу `int` (например, при сложении двух значений типа `char` они оба сначала приводятся к `int`).
4. Если оба аргумента целочисленные, то аргумент с меньшим типом приводится к типу второго аргумента (например, при сложении `long` и `int` значение типа `int` приводится к `long`).
5. Если оба аргумента целочисленные и имеют тип одного размера, то предпочтение отдаётся беззнаковому типу (например, при сложении `int` и `unsigned int` значение типа `int` приводится к `unsigned int`).

Следствия неявных преобразований типов

1. Следите за тем, какие типы участвуют в выражении, от этого может зависеть его значение.
2. Не стоит использовать целочисленные типы меньше `int` в арифметических выражениях, они всё равно будут приведены к `int`.
3. **Не стоит смешивать `unsigned` и `signed`** типы в одном выражении, это может привести к неприятным последствиям.

Для иллюстрации последнего следствия давайте рассмотрим следующий пример:

```
unsigned from = 100;
unsigned to = 0;
for (int i = from; i >= to; --i) {  ....  }
```

Инструкции

- Выполнение состоит из последовательности *инструкций*.
- Инструкции выполняются одна за другой.
- Порядок вычислений внутри инструкций неопределен.

```
/* unspecified behavior */
int i = 10;
i = (i+=6) + (i * 4);
```

- Блоки имеют вложенную область видимости:

```
int k = 10;
{
    int k = 5 * i; // не видна за пределами блока
    i = (k += 5) + 5;
}
k = k + 1;
```

Условные операторы

- Оператор **if**:

```
int d = b * b - 4 * a * c;
if (d > 0)
{
    roots = 2;
}
else if (d == 0)
{
    roots = 1;
}
else
{
    roots = 0;
}
```

- Тернарный условный оператор:

```
int roots = 0;
if (d >= 0)
    roots = (d > 0) ? 2 : 1;
```

Циклы

- Цикл **while**:

```
int squares = 0;
int k = 0;
while (k < 0)
{
    squares += k * k;
    k = k + 1;
}
```

- Цикл **for**:

```
for (int k = 0; k < 10; k = k + 1)
{
    squares += k * k;
}
```

- Для выхода из цикла используется оператор **break**.

Цикл do-while

В C++ существует вариация цикла `while`, которая называется `do-while`. В отличие от обычного `while` в `do-while` условие проверяется не до, а *после* итерации. Т.е. такой цикл всегда имеет хотя бы одну итерацию.

```
int i = 10;
int sum = 0;
while (i < 10) {
    sum += i;
}
// sum = 0
```

```
int i = 10;
int sum = 0;
do {
    sum += i;
} while(i < 10);
// sum = 10
```

Управление циклами

Ранее был упомянут оператор **break**, который используется для выхода из цикла. Рассмотрим его действие на следующем примере.

```
int a = 323;
int b = 2;
while ( b <= a ) {
    if ( a % b == 0 )
    break; // выйти из цикла
    b = b + 1;
}
```

В данном случае b будет равен 17, т.к. $323 = 17 \times 19$.

Ещё один оператор, который можно использовать с циклами — это оператор **continue**. Оператор **continue** прерывает текущую итерацию цикла и переходит к следующей.

```
int sum = 0;
for ( int i = 1; i <= 100; ++i ) {
    if ( (i % 17 == 0) || (i % 19 == 0) )
    continue; // перейти к следующей итерации
    sum += i;
}
```

Потоки ввода и вывода

Вывод различных типов в C++

```
#include <iostream>

int main()
{
    int i = 42;
    double d = 3.14;
    const char *s = "C-style string";

    std::cout << "This is an integer " << i << "\n";
    std::cout << "This is a double " << d << "\n";
    std::cout << "This is a \"" << s << "\"\n";

    return 0;
}
```

Потоки ввода и вывода

Вывод и ввод в C++

```
#include <iostream>

int main()
{
    int i = 42;
    double d = 3.14;

    std::cout << "Enter an integer and a double:\n";
    std::cin >> i >> d;
    std::cout << "Your input is " << i << ", " << d <<
    ↪   "\n";

    return 0;
}
```

Посимвольный ввод

```
char c = '\0';
while (cin.get(c)) { // на каждой итерации считываем
 ↪  один символ в переменную c
    /* здесь можно пользоваться значением прочитанным
     ↪  в переменную c */
    if (c != 'a')
    cout << c; // выводим символ, если он не равен 'a'
}
```

Функции

- В сигнатуре функции указывается тип возвращаемого значений и типы параметров.
- Ключевое слово return возвращает значение.

```
double square ( double x ) {
    return x * x ;
}
```

- Переменные, определённые внутри функций, — *локальные*.
- Функция может возвращать **void**
- Параметры передаются по значению (копируются).

```
void strange ( double x , double y ) {
    x = y ;
}
```

Макросы

- Макросами в C++ называют инструкции препроцессора.
- Препроцессор C++ является самостоятельным языком, работающим с произвольными строками.
- Макросы можно использовать для определения функций:

```
int max1 ( int x , int y ) {
    return x > y ? x : y ;
}
#define max2 (x , y ) x > y ? x : y
a = b + max2 (c , d ); // b + c > d ? c : d;
```

- Препроцессор “не знает” про синтаксис C++

Макросы

- Параметры макросов нужно оборачивать в скобки:

```
#define max3 (x , y ) (( x ) > ( y ) ? (x ) : ( y
↪   ))
```

- Это не избавляет от всех проблем:

```
int a = 1;
int b = 1;
int c = max3 (++ a , b );
// c = ((++a) > (b) ? (++a) : (b))
```

- Определять функции через макросы — плохая идея.
- Макросы можно использовать для условной компиляции:

```
#ifdef DEBUG
// дополнительные проверки
#endif
```

Макросы

Предположим, что в программе определён макрос sqr.

```
#define sqr(x) x * x
```

Какое значение будет иметь следующее выражение?

```
sqr(3 + 0)
```

Ввод-вывод

- Будем использовать библиотеку <iostream>

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

- Ввод:

```
int a = 0;
int b = 0;
cin >> a >> b ;
```

- Вывод:

```
cout << " a + b = " << ( a + b ) << endl;
```

Простая программа

```
#include <iostream>
using namespace std ;
int main ()
{
    int a = 0;
    int b = 0;
    cout << " Enter a and b : " ;
    cin >> a >> b ;
    cout << " a + b = " << ( a + b ) << endl ;
    return 0;
}
```