

# Программирование на языке C++

## Лекция 1

История языка C++

Александр Смаль

## Язык С

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

## Особенности С

- **Эффективность.**

Язык С позволяет писать программы, которые напрямую работают с железом.

- **Стандартизированность.**

Спецификация языка С является международным стандартом.

- **Относительная простота.**

Стандарт языка С занимает 230 страниц (против 670 для Java и 1340 для C++).

## Создание C++

- Разрабатывается с начала 1980-х годов.
- Создатель — сотрудник Bell Labs *Бьёрн Страуструп*.
- Изначально это было расширение языка С 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, перерабатывающий исходный код “C с классами” в исходный код на С.

## Развитие C++

- К 1983 году в язык было добавлено много новых возможностей (виртуальные функции, перегрузка функций и операторов, ссылки, константы, ...)
- Получившийся язык перестал быть просто дополненной версией классического С и был переименован из “С с классами” в C++.
- Имя языка, получившееся в итоге, происходит от оператора унарного постфиксного инкремента С '++' (увеличение значения переменной на единицу).
- Язык также не был назван D, поскольку “является расширением С и не пытается устраниить проблемы путём удаления элементов С”.
- 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 недочёты 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.
- В данный момент ведётся одновременная работа над двумя версиями стандарта: C++14 и C++17.

## Совместимость С и С++

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

# Программирование на языке C++

## Лекция 1

Характеристики языка C++

Александр Смаль

# Характеристики языка C++

## Характеристики C++:

- сложный,
- мультипарадигмальный,
- эффективный,
- низкоуровневый,
- компилируемый,
- статически типизированный.

## Сложность

- Описание стандарта занимает более 1300 страниц текста.
- Нет никакой возможности рассказать “весь C++” в рамках одного, пусть даже очень большого курса.
- В C++ программисту позволено очень многое, и это влечёт за собой большую ответственность.
- На плечи программиста ложится много дополнительной работы:
  - проверка корректности данных,
  - управление памятью,
  - обработка низкоуровневых ошибок.

# Мультипарадигмальный

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

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

## Эффективный

Одна из фундаментальных идей языков С и С++ — 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 С++ основным языком для разработки приложений с компьютерной графикой (к примеру, игры).

## Низкоуровневый

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.

## Компилируемый

C++ является компилируемым языком программирования.

Для того, чтобы запустить программу на C++, её 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 идёт работа с последовательностями байт.

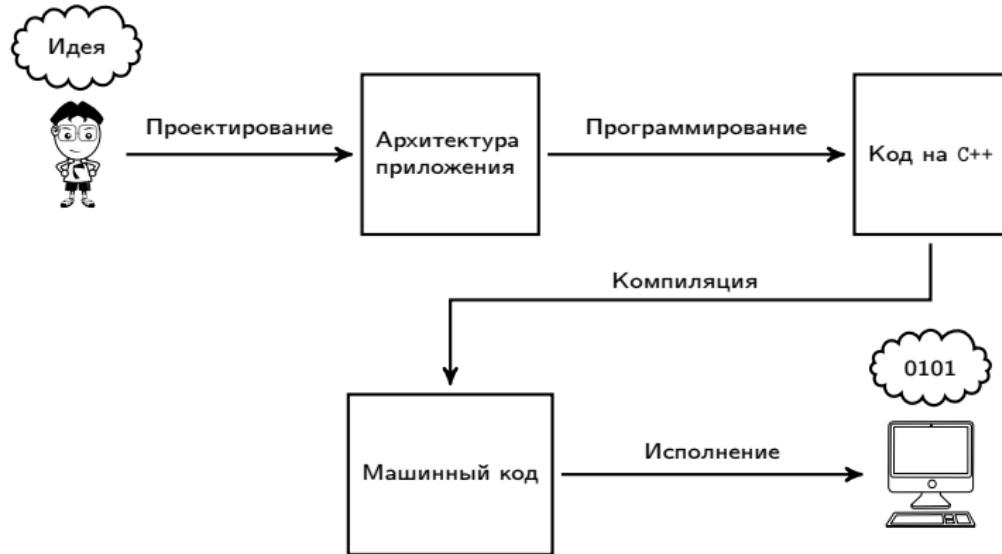
# Программирование на языке C++

## Лекция 1

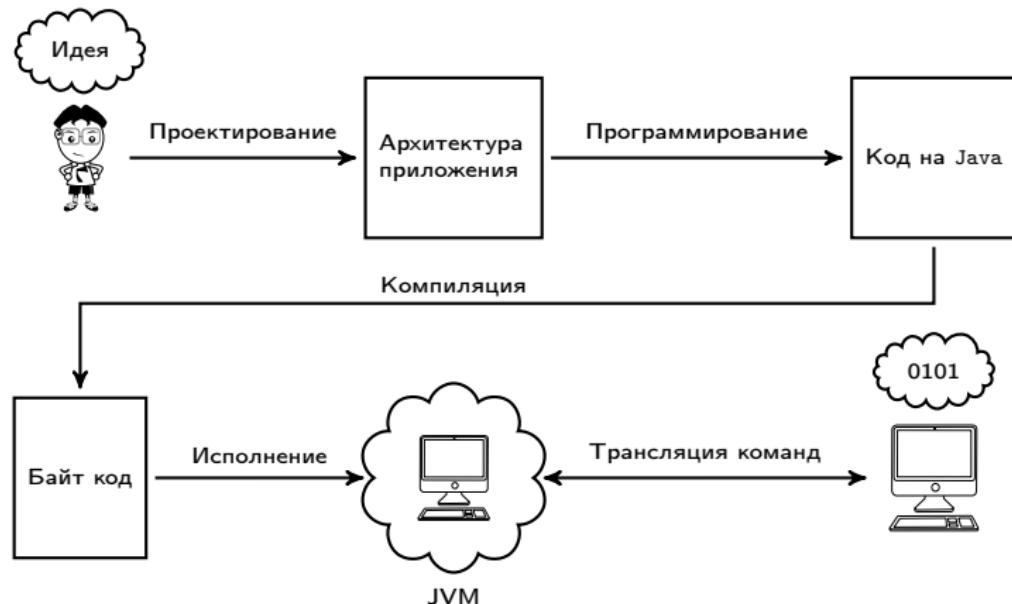
Зачем нужен компилятор?

Александр Смаль

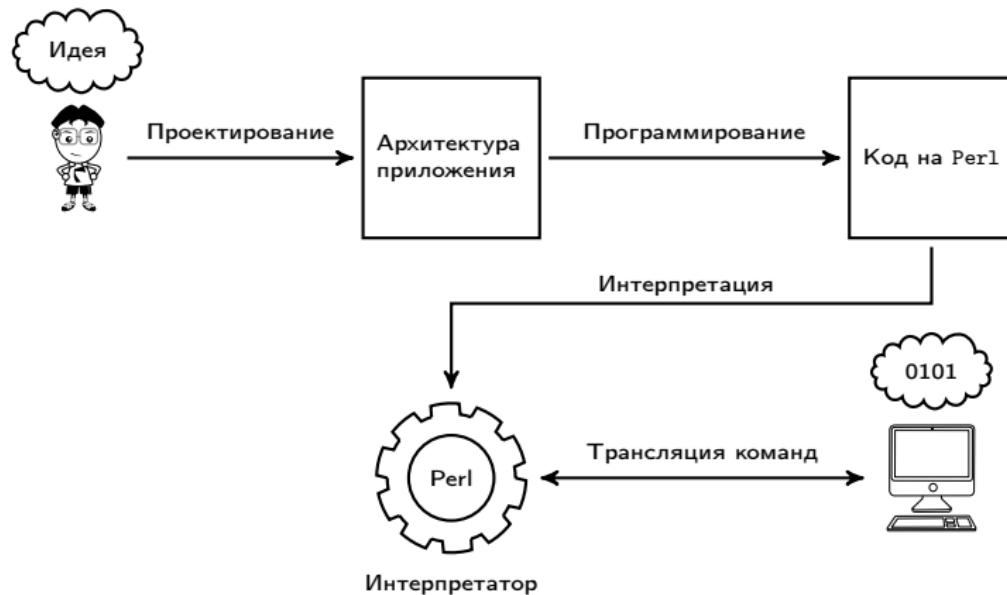
# Что такое компиляция?



# Что такое компиляция?



# Что такое интерпретация?



# 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 изменения в программу — нужно перекомпилировать заново.

**Важно:** компиляция — преобразование одностороннее, нельзя восстановить исходный код.

# Программирование на языке C++

## Лекция 1

Структура кода на C++

Александр Смаль

# 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, иногда .C),
2. заголовочные файлы (расширение .hpp или .h).

# Заголовочные файлы

- Файл `foo.cpp`:

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

- Файл `bar.cpp`:

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

Компиляция этих файлов выдаст ошибку.

# Заголовочные файлы

- Файл `foo.cpp`:

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

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

- Файл `bar.cpp`:

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

# Заголовочные файлы

Предположим, что мы изменили функцию bar.

- Файл foo.cpp:

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

- Файл bar.cpp:

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

Данный код некорректен — объявление отличается от определения. (Неопределённое поведение.)

# Заголовочные файлы

Добавим заголовочный файл bar.hpp.

- Файл foo.cpp:

```
#include "bar.hpp"

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

- Файл bar.cpp:

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

- Файл bar.hpp:

```
int bar();
```

## 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"

void foo();
```

## Стражи включения

Это можно исправить двумя способами:

- (наиболее переносимо) Файл bar.hpp:

```
#ifndef BAR_HPP
#define BAR_HPP

int bar();
#endif
```

- (наиболее просто) Файл bar.hpp:

```
#pragma once

int bar();
```

**Резюме:** .cpp — для определений, .hpp — для объявлений.

# Программирование на языке C++

## Лекция 1

Как компилируется программа на C++?

Александр Смаль

## 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.
- При этом происходит подстановка адресов функций в места их вызова.

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

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

- По каждому объектному файлу строится таблица всех функций, которые в нём 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, double) {}
```

в `_Z3foooid`.

- Аналогично функциям в линковке нуждаются глобальные переменные.

## Этап 3: линковка (компоновка)

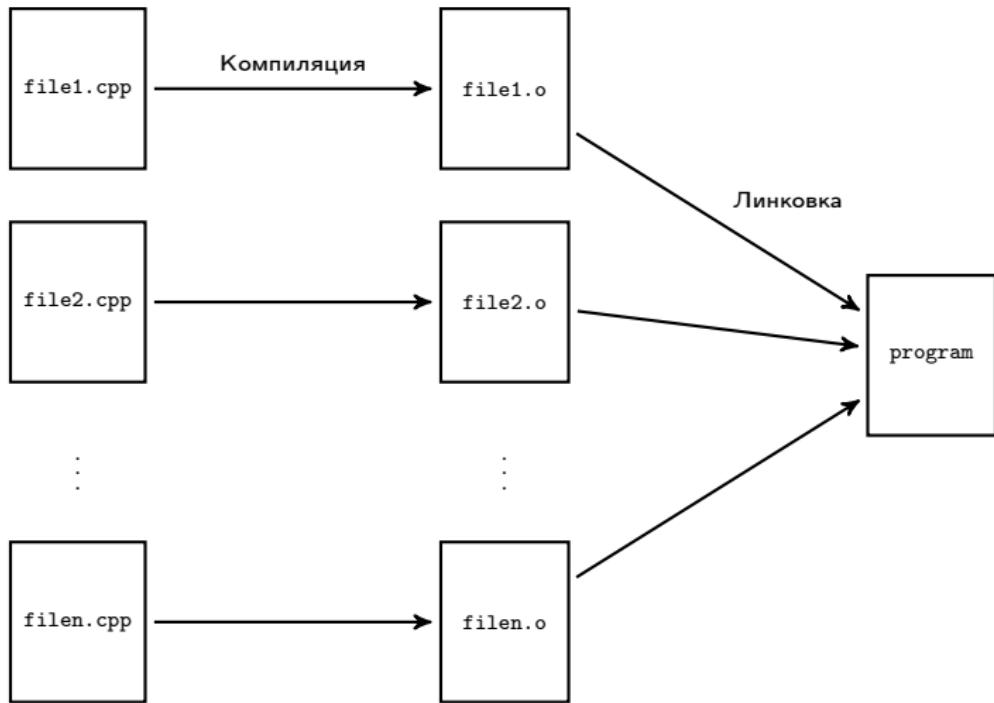
- Точка входа — функция, вызываемая при запуске программы. По умолчанию — это функция `main`:

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

или

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

# Общая схема



# Программирование на языке C++

## Лекция 1

Введение в синтаксис C++

Александр Смаль

# Типы данных

- Целочисленные:
  1. `char` (символьный тип данных)
  2. `short int`
  3. `int`
  4. `long int`

Могут быть беззнаковыми (`unsigned`).

- $-2^{n-1} \dots (2^{n-1} - 1)$  ( $n$  — 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. 'а' — код буквы 'а', тип `char`,
  2. 42 — все целые числа по умолчанию типа `int`,
  3. 1234567890L — суффикс 'L' соответствует типу `long`,
  4. 1703U — суффикс 'U' соответствует типу `unsigned int`,
  5. 2128506UL — соответствует типу `unsigned long`.
- Числа с плавающей точкой:
  1. 3.14 — все числа с точкой по умолчанию типа `double`,
  2. 2.71F — суффикс 'F' соответствует типу `float`,
  3. 3.0E8 — соответствует  $3.0 \cdot 10^8$ .
- `true` и `false` — значения типа `bool`.
- Строки задаются в двойных кавычках: "Text 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 = 123123;

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

- Нужно всегда инициализировать переменные.
- Нельзя определить переменную пустого типа `void`.

# Операции

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

# Инструкции

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

```
/* unspecified behavior */
int i = 10;
i = (i += 5) + (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 < 10 ) {
    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`.

# 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 обрамлять 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
```

# 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;
}
```

# Программирование на языке C++

## Лекция 1

Введение в синтаксис C++

Александр Смаль

# Типы данных

- Целочисленные:
  1. `char` (символьный тип данных)
  2. `short int`
  3. `int`
  4. `long int`

Могут быть беззнаковыми (`unsigned`).

- $-2^{n-1} \dots (2^{n-1} - 1)$  ( $n$  — 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. 'а' — код буквы 'а', тип `char`,
  2. 42 — все целые числа по умолчанию типа `int`,
  3. 1234567890L — суффикс 'L' соответствует типу `long`,
  4. 1703U — суффикс 'U' соответствует типу `unsigned int`,
  5. 2128506UL — соответствует типу `unsigned long`.
- Числа с плавающей точкой:
  1. 3.14 — все числа с точкой по умолчанию типа `double`,
  2. 2.71F — суффикс 'F' соответствует типу `float`,
  3. 3.0E8 — соответствует  $3.0 \cdot 10^8$ .
- `true` и `false` — значения типа `bool`.
- Строки задаются в двойных кавычках: "Text 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 = 123123;

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

- Нужно всегда инициализировать переменные.
- Нельзя определить переменную пустого типа `void`.

# Операции

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

# Инструкции

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

```
/* unspecified behavior */
int i = 10;
i = (i += 5) + (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 < 10 ) {
    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`.

# 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 обрамлять 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
```

# 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;
}
```

# Программирование на языке C++

## Лекция 2

Как выполняются программы на C++

Александр Смаль

# Архитектура фон Неймана

Современных компьютеры построены по принципам архитектуры фон Неймана:

- 1. Принцип однородности памяти.**

Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы.

- 2. Принцип адресности.**

Память состоит из пронумерованных ячеек.

- 3. Принцип программного управления.**

Все вычисления представляются в виде последовательности команд.

- 4. Принцип двоичного кодирования.**

Вся информация (данные и команды) кодируются двоичными числами.

## Сегментация памяти

- Оперативная память, используемая в программе на C++, разделена на области двух типов:
  1. сегменты данных,
  2. сегменты кода (текстовые сегменты).
- В сегментах кода содержится код программы.
- В сегментах данных располагаются данные программы (значения переменных, массивы и пр.).
- При запуске программы выделяются два сегмента данных:
  1. сегмент глобальных данных,
  2. стек.
- В процессе работы программы могут выделяться и освобождаться дополнительные сегменты памяти.
- Обращения к адресу вне выделенных сегментов — ошибка времени выполнения (access violation, segmentation fault).

# Как выполняется программа?

- Каждой функции в скомпилированном коде соответствует отдельная секция.
- Адрес начала такой секции — это адрес функции.
- Телу функции соответствует последовательность команд процессора.
- Работа с данными происходит на уровне байт, информация о типах отсутствует.
- В процессе выполнения адрес следующей инструкции хранится в специальном регистре процессора IP (Instruction Pointer).
- Команды выполняются последовательно, пока не встретится специальная команда (например, условный переход или вызов функции), которая изменит IP.

## Ещё раз о линковке

- На этапе компиляции объектных файлов в места вызова функций подставляются имена функций.
- На этапе линковки в места вызова вместо имён функций подставляются их адреса.
- Ошибки линковки:
  1. `undefined reference`  
Функция имеет объявление, но не имеет тела.
  2. `multiple definition`  
Функция имеет два или более определений.
- Наиболее распространённый способ получить `multiple definition` — определить функцию в заголовочном файле, который включён в несколько .cpp файлов.

# Программирование на языке C++

## Лекция 2

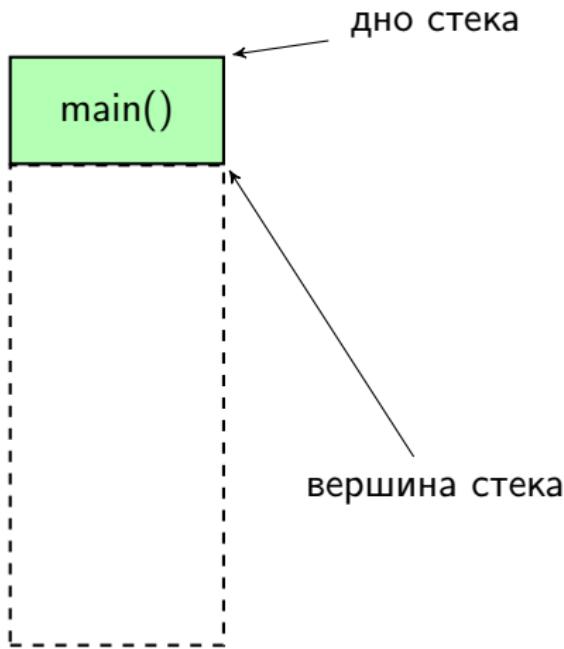
Стек вызовов

Александр Смаль

## Стек вызовов

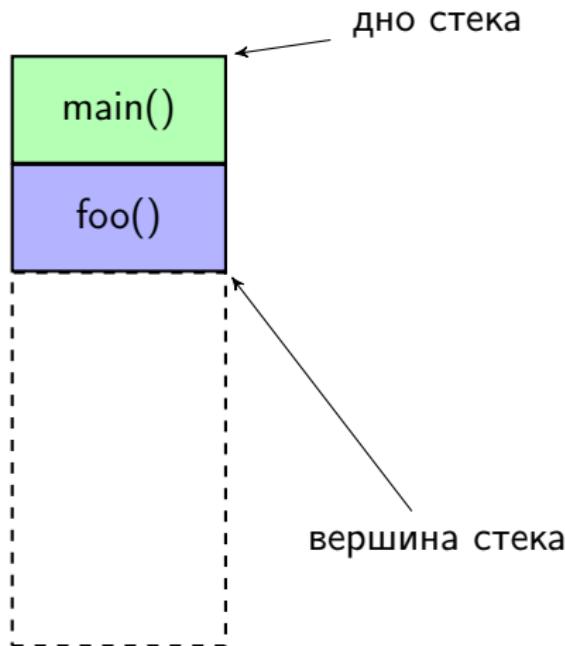
- Стек вызовов — это сегмент данных, используемый для хранения локальных переменных и временных значений.
- Не стоит путать стек с одноимённой структурой данных, у стека в C++ можно обратиться к произвольной ячейке.
- Стек выделяется при запуске программы.
- Стек обычно небольшой по размеру (4Мб).
- Функции хранят свои локальные переменные на стеке.
- При выходе из функции соответствующая область стека объявляется свободной.
- Промежуточные значения, возникающие при вычислении сложных выражений, также хранятся на стеке.

# Устройство стека



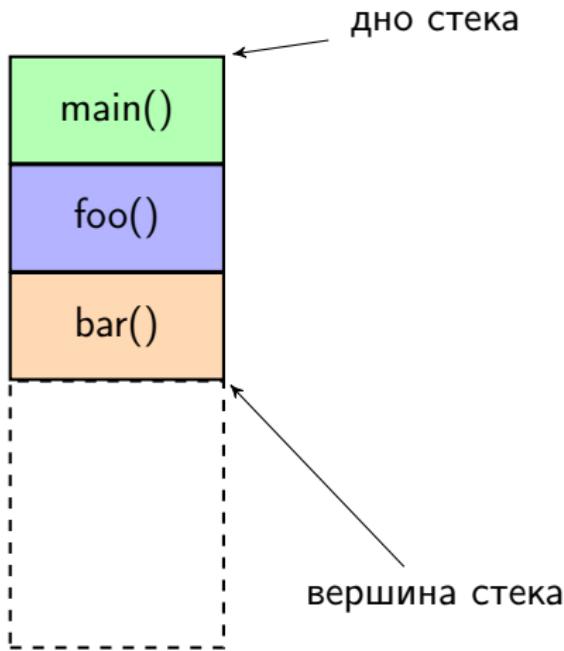
```
void bar( ) {  
    int c;  
}  
  
void foo( ) {  
    int b = 3;  
    bar();  
}  
  
int main( ) {  
    int a = 3;  
    foo();  
    bar();  
  
    return 0;  
}
```

# Устройство стека



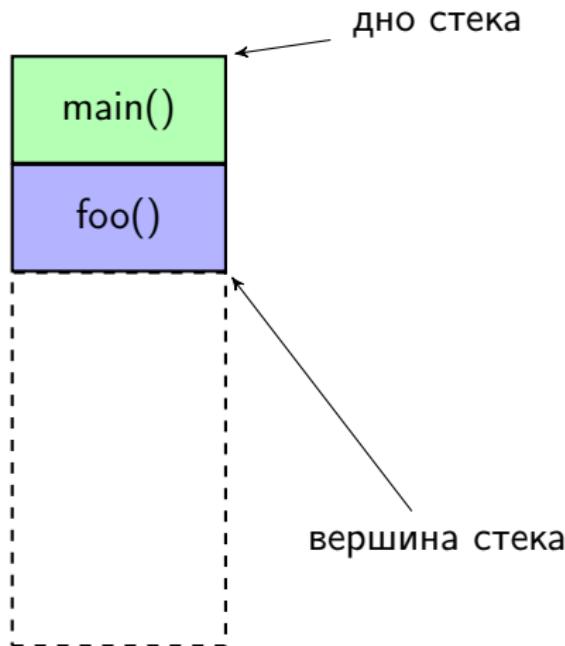
```
void bar( ) {  
    int c;  
}  
  
void foo( ) {  
    int b = 3;  
    bar();  
}  
  
int main( ) {  
    int a = 3;  
    foo();  
    bar();  
  
    return 0;  
}
```

# Устройство стека



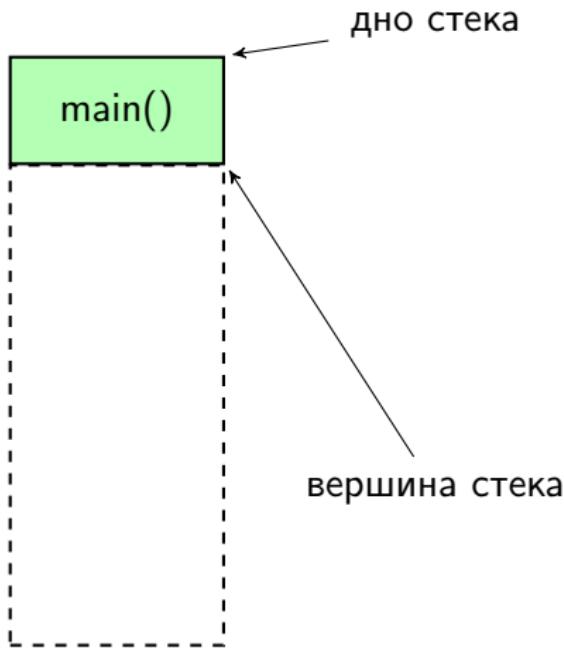
```
void bar( ) {  
    int c;  
}  
  
void foo( ) {  
    int b = 3;  
    bar();  
}  
  
int main( ) {  
    int a = 3;  
    foo();  
    bar();  
  
    return 0;  
}
```

# Устройство стека



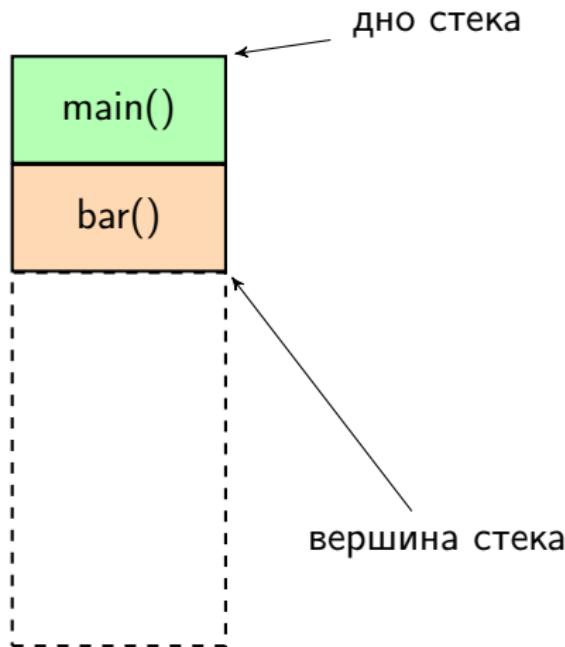
```
void bar( ) {  
    int c;  
}  
  
void foo( ) {  
    int b = 3;  
    bar();  
}  
  
int main( ) {  
    int a = 3;  
    foo();  
    bar();  
  
    return 0;  
}
```

# Устройство стека



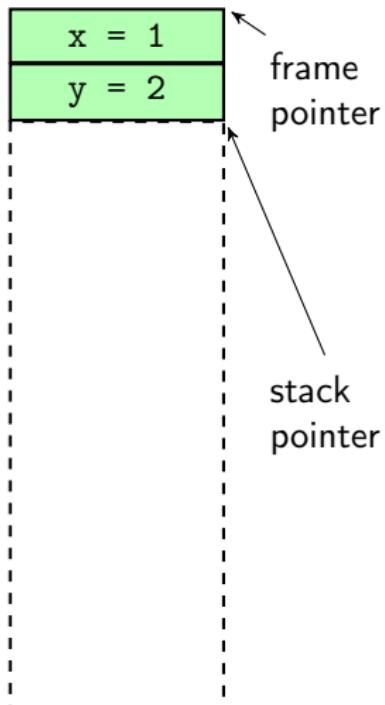
```
void bar( ) {  
    int c;  
}  
  
void foo( ) {  
    int b = 3;  
    bar();  
}  
  
int main( ) {  
    int a = 3;  
    foo();  
    bar();  
  
    return 0;  
}
```

# Устройство стека



```
void bar( ) {  
    int c;  
}  
  
void foo( ) {  
    int b = 3;  
    bar();  
}  
  
int main( ) {  
    int a = 3;  
    foo();  
    bar();  
  
    return 0;  
}
```

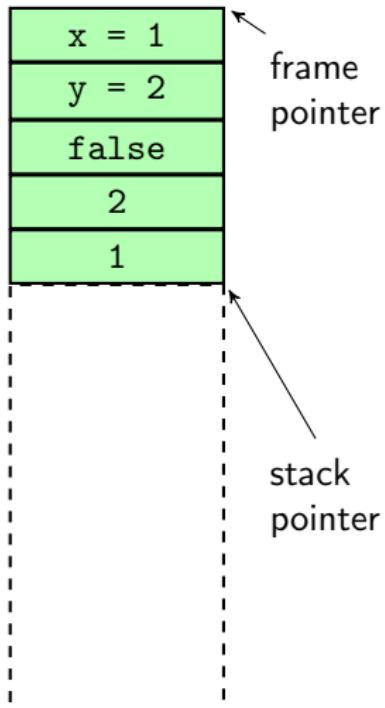
## Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main()
{
    int x = 1;
    int y = 2;
    x = foo(x, y, false);
    cout << x;
    return 0;
}
```

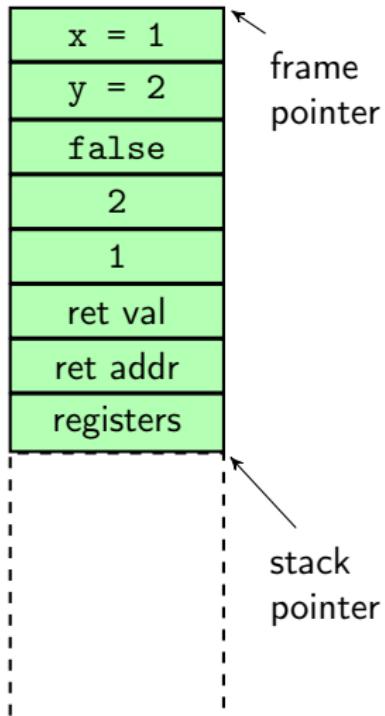
## Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

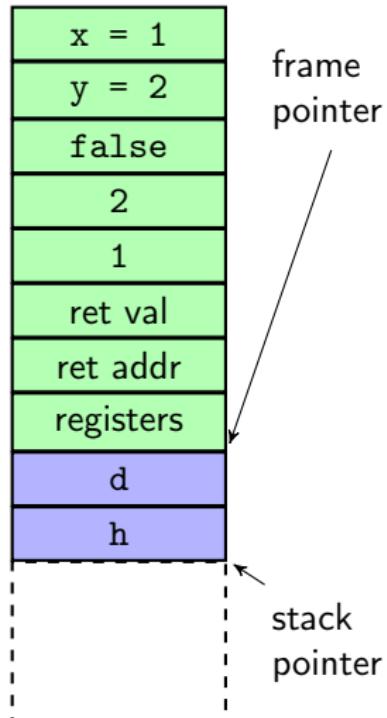
## Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

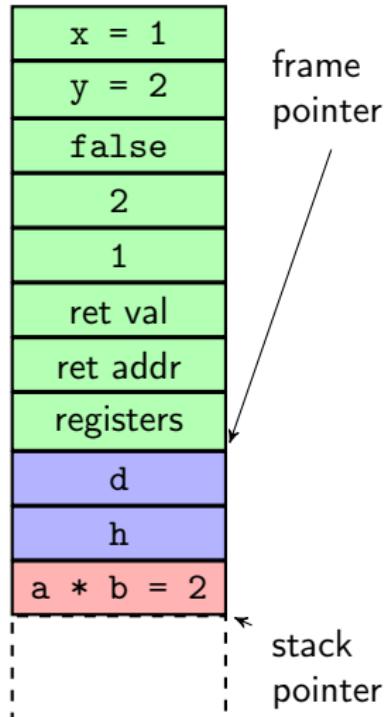
# Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

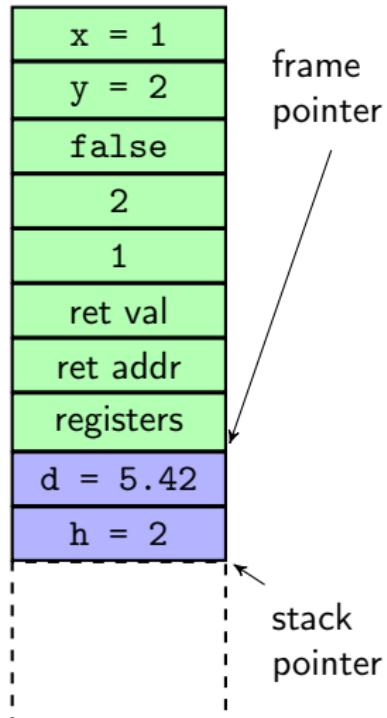
## Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

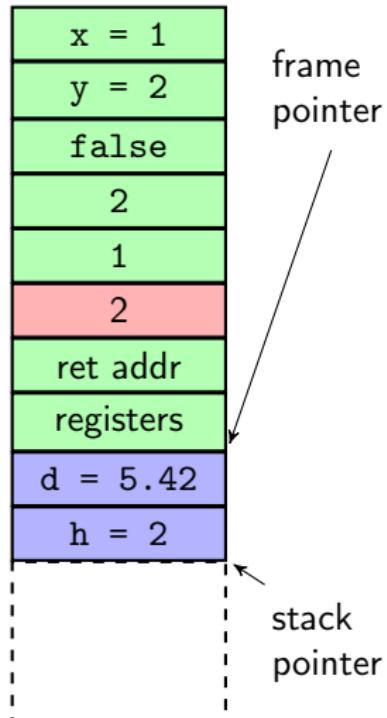
# Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

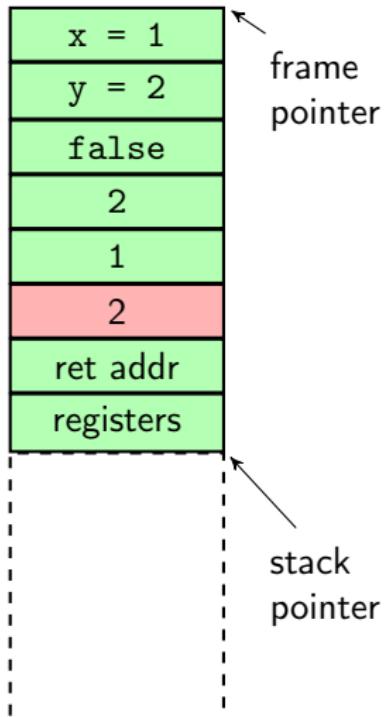
# Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

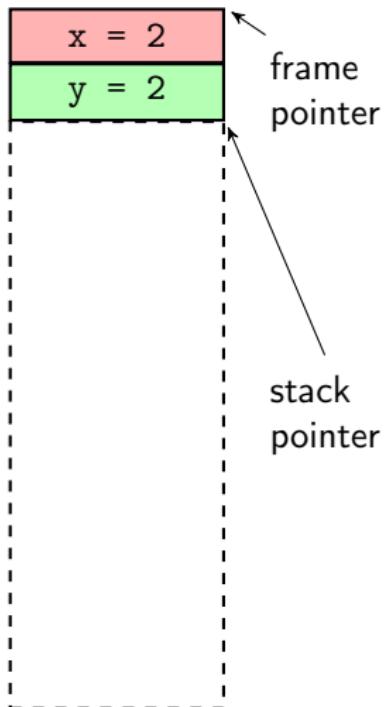
## Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main( )
{
    int x = 1;
    int y = 2;
    x = foo (x, y, false);
    cout << x;
    return 0;
}
```

## Вызов функции



```
int foo(int a, int b, bool c)
{
    double d = a * b * 2.71;
    int     h = c ? d : d / 2;
    return h;
}

int main()
{
    int x = 1;
    int y = 2;
    x = foo(x, y, false);
    cout << x;
    return 0;
}
```

## Вызов функции

- При вызове функции на стек складываются:
  1. аргументы функции,
  2. адрес возврата,
  3. значение frame pointer и регистров процессора.
- Кроме этого на стеке резервируется место под возвращаемое значение.
- Параметры передаются в обратном порядке, что позволяет реализовать функции с переменным числом аргументов.
- Адресация локальных переменных функции и аргументов функции происходит относительно frame pointer.
- Конкретный процесс вызова зависит от используемых соглашений (cdecl, stdcall, fastcall, thiscall).

# Программирование на языке C++

## Лекция 2

Указатели и массивы

Александр Смаль

# Указатели

- Указатель — это переменная, хранящая адрес некоторой ячейки памяти.
- Указатели являются типизированными.

```
int      i = 3; // переменная типа int
int * p = 0; // указатель на переменную типа int
```

- Нулевому указателю (к которому присвоено значение 0) не соответствует никакая ячейка памяти.
- Оператор взятия адреса переменной &.
- Оператор разыменования \*.

```
p = &i; // указатель p указывает на переменную i
*p = 10; // изменяется ячейка по адресу p, т.е. i
```

## Передача параметров по указателю

Рассмотрим функцию, меняющую параметры местами:

```
void swap (int a, int b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int k = 10, m = 20;
    swap (k, m);
    cout << k << ', ' << m << endl; // 10 20
    return 0;
}
```

swap изменяет локальные копии переменных k и m.

## Передача параметров по указателю

Вместо значений типа `int` будем передавать указатели.

```
void swap (int * a, int * b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int main() {
    int k = 10, m = 20;
    swap (&k, &m);
    cout << k << ', ' << m << endl; // 20 10
    return 0;
}
```

`swap` изменяет переменные `k` и `m` по указателям на них.

# Массивы

- Массив — это набор однотипных элементов, расположенных в памяти друг за другом, доступ к которым осуществляется по индексу.
- C++ позволяет определять массивы на стеке.

```
// массив 1 2 3 4 5 0 0 0 0 0  
int m[10] = {1, 2, 3, 4, 5};
```

- Индексация массива начинается с 0, последний элемент массива длины n имеет индекс n - 1.

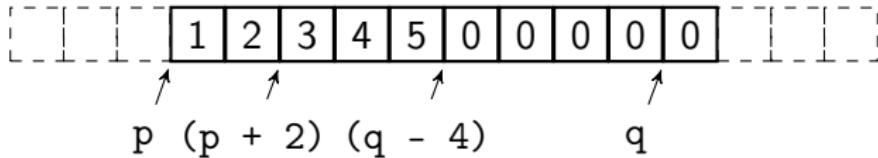
```
for (int i = 0; i < 10; ++i)  
    cout << m[i] << ' ';  
cout << endl;
```

# Связь массивов и указателей

- Указатели позволяют передвигаться по массивам.
- Для этого используется арифметика указателей:

```
int m[10] = {1, 2, 3, 4, 5};  
int * p = &m[0]; // адрес начала массива  
int * q = &m[9]; // адрес последнего элемента
```

- $(p + k)$  — сдвиг на  $k$  ячеек типа `int` вправо.
- $(p - k)$  — сдвиг на  $k$  ячеек типа `int` влево.
- $(q - p)$  — количество ячеек между указателями.
- $p[k]$  эквивалентно  $*(p + k)$ .



# Примеры

Заполнение массива:

```
int m[10] = {}; // изначально заполнен нулями
//           &m[0]           &m[9]
for (int * p = m ; p <= m + 9; ++p )
    *p = (p - m) + 1;
// Массив заполнен числами от 1 до 10
```

Передача массива в функцию:

```
int max_element (int * m, int size) {
    int max = *m;
    for (int i = 1; i < size; ++i)
        if (m[i] > max)
            max = m[i];
    return max;
}
```

# Программирование на языке C++

## Лекция 2

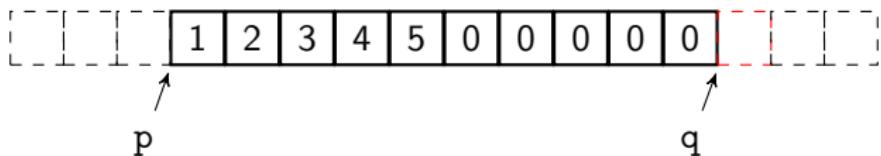
Использование указателей

Александр Смаль

## Два способа передачи массива

Функция для поиска элемента в массиве:

```
bool contains(int * m, int size, int value) {  
    for (int i = 0; i != size; ++i)  
        if (m[i] == value)  
            return true;  
    return false;  
}  
  
bool contains(int * p, int * q, int value) {  
    for (; p != q; ++p)  
        if (*p == value)  
            return true;  
    return false;  
}
```



## Возрат указателя из функции

Функция для поиска максимума в массиве:

```
int max_element (int * p, int * q) {
    int max = *p;
    for ( ; p != q; ++p)
        if (*p > max)
            max = *p;

    return max;
}
```

```
int m[10] = {...};
int max = max_element(m, m + 10);
cout << "Maximum = " << max << endl;
```

## Возрат указателя из функции

Функция для поиска максимума в массиве:

```
int * max_element (int * p, int * q) {
    int * pmax = p;
    for (; p != q; ++p)
        if (*p > *pmax)
            pmax = p;

    return pmax;
}
```

```
int m[10] = {...};
int * pmax = max_element(m, m + 10);
cout << "Maximum = " << *pmax << endl;
```

## Возрят значения через указатель

Функция для поиска максимума в массиве:

```
bool max_element (int * p, int * q, int * res) {
    if (p == q)
        return false;
    *res = *p;
    for (; p != q; ++p)
        if (*p > *res)
            *res = *p;
    return true;
}
```

```
int m[10] = {...};
int max = 0;
if (max_element(m, m + 10, &max))
    cout << "Maximum = " << max << endl;
```

## Возрат значения через указатель на указатель

Функция для поиска максимума в массиве:

```
bool max_element (int * p, int * q, int ** res) {
    if (p == q)
        return false;
    *res = p;
    for (; p != q; ++p)
        if (*p > **res)
            *res = p;
    return true;
}
```

```
int m[10] = {...};
int * pmax = 0;
if (max_element(m, m + 10, &pmax))
    cout << "Maximum = " << *pmax << endl;
```

# Программирование на языке C++

## Лекция 2

Ссылки

Александр Смаль

## Недостатки указателей

- Использование указателей синтаксически загрязняет код и усложняет его понимание. (Приходится использовать операторы `*` и `&`.)
- Указатели могут быть неинициализированными (некорректный код).
- Указатель может быть нулевым (корректный код), а значит указатель нужно проверять на равенство нулю.
- Арифметика указателей может сделать из корректного указателя некорректный (легко промахнуться).

## Ссылки

- Для того, чтобы исправить некоторые недостатки указателей, в C++ введены ссылки.
- Ссылки являются “красивой обёрткой” над указателями:

```
void swap (int & a, int & b) {
    int t = b;
    b = a;
    a = t;
}

int main() {
    int k = 10, m = 20;
    swap (k, m);
    cout << k << ' ' << m << endl; // 20 10
    return 0;
}
```

## Различия ссылок и указателей

- Ссылка не может быть неинициализированной.

```
int * p; // OK  
int & l; // ошибка
```

- У ссылки нет нулевого значения.

```
int * p = 0; // OK  
int & l = 0; // ошибка
```

- Ссылку нельзя переинициализировать.

```
int a = 10, b = 20;  
int * p = &a; // p указывает на a  
p = &b; // p указывает на b  
int & l = a; // l ссылается на a  
l = b; // a присваивается значение b
```

## Различия ссылок и указателей

- Нельзя получить адрес ссылки или ссылку на ссылку.

```
int a = 10;
int * p = &a;    // p указывает на a
int ** pp = &p; // pp указывает на переменную p
int & l = a;    // l ссылается на a
int * pl = &l; // pl указывает на переменную a
int && ll = l; // ошибка
```

- Нельзя создавать массивы ссылок.

```
int * mp[10] = {};// массив указателей на int
int & ml[10] = {};// ошибка
```

- Для ссылок нет арифметики.

## lvalue и rvalue

- Выражения в C++ можно разделить на два типа:
  - lvalue** — выражения, значения которых являются *ссылкой* на переменную/элемент массива, а значит могут быть указаны слева от оператора `=`.
  - rvalue** — выражения, значения которых являются временными и не соответствуют никакой переменной/элементу массива.
- Указатели и ссылки могут указывать только на lvalue.

```
int a = 10, b = 20;
int m[10] = {1,2,3,4,5,5,4,3,2,1};
int & l1 = a;           // OK
int & l2 = a + b;      // ошибка
int & l3 = *(m + a / 2); // OK
int & l4 = *(m + a / 2) + 1; // ошибка
int & l5 = (a + b > 10) ? a : b; // OK
```

## Время жизни переменной

Следует следить за временем жизни переменных.

```
int * foo() {
    int a = 10;
    return &a;
}
```

```
int & bar() {
    int b = 20;
    return b;
}
```

```
int * p = foo();
int & l = bar();
```

# Программирование на языке C++

## Лекция 2

Динамическая память

Александр Смаль

## Зачем нужна динамическая память?

- Стек программы ограничен. Он не предназначен для хранения больших объемов данных.

```
// Не умещается на стек  
double m[10000000] = {};// 80 Mb
```

- Время жизни локальных переменных ограничено временем работы функции.
- Динамическая память выделяется в сегменте данных.
- Структура, отвечающая за выделение дополнительной памяти, называется **кучей** (не нужно путать с одноимённой структурой данных).
- Выделение и освобождение памяти **управляется вручную**.

## Выделение памяти в стиле С

- Стандартная библиотека `cstdlib` предоставляет четыре функции для управления памятью:

```
void * malloc (size_t size);
void   free   (void * ptr);
void * calloc (size_t nmemb, size_t size);
void * realloc(void * ptr, size_t size);
```

- `size_t` — специальный целочисленный беззнаковый тип, может вместить в себя размер любого типа в байтах.
- Тип `size_t` используется для указания размеров типов данных, для индексации массивов и пр.
- `void *` — это указатель на нетипизированную память (раньше для этого использовалось `char *`).

## Выделение памяти в стиле С

- Функции для управления памятью в стиле С:

```
void * malloc (size_t size);
void * calloc (size_t nmemb, size_t size);
void * realloc(void * ptr, size_t size);
void free   (void * ptr);
```

- `malloc` — выделяет область памяти размера  $\geq \text{size}$ .  
Данные не инициализируются.
- `calloc` — выделяет массив из `nmemb` размера `size`.  
Данные инициализируются нулём.
- `realloc` — изменяет размер области памяти по указателю `ptr` на `size` (если возможно, то это делается на месте).
- `free` — освобождает область памяти, ранее выделенную одной из функций `malloc/calloc/realloc`.

## Выделение памяти в стиле С

- Для указания размера типа используется оператор `sizeof`.

```
// создание массива из 1000 int
int * m = (int *)malloc(1000 * sizeof(int));
m[10] = 10;

// изменение размера массива до 2000
m = (int *)realloc(m, 2000 * sizeof(int));

// освобождение массива
free(m);

// создание массива нулей
m = (int *)calloc(3000, sizeof(int));

free(m);
m = 0;
```

## Выделение памяти в стиле C++

- Язык C++ предоставляет два набора операторов для выделения памяти:
  1. `new` и `delete` — для одиночных значений,
  2. `new []` и `delete []` — для массивов.
- Версия оператора `delete` должна соответствовать версии оператора `new`.

```
// выделение памяти под один int со значением 5
int * m = new int(5);
delete m; // освобождение памяти

// создание массива значений типа int
m = new int[1000];
delete [] m; // освобождение памяти
```

## Типичные проблемы при работе с памятью

- Проблемы производительности: создание переменной на стеке намного “дешевле” выделения для неё динамической памяти.
- Проблема фрагментации: выделение большого количества небольших сегментов способствует фрагментации памяти.
- Утечки памяти:

```
// создание массива из 1000 int
int * m = new int[1000];

// создание массива из 2000 int
m = new int[2000]; // утечка памяти

// Не вызван delete [] m, утечка памяти
```

## Типичные проблемы при работе с памятью

- Неправильное освобождение памяти.

```
int * m1 = new int[1000];
delete m1; // должно быть delete [] m1

int * p = new int(0);
free(p); // совмещение функций C++ и С

int * q1 = (int *)malloc(sizeof(int));
free(q1);
free(q1); // двойное удаление

int * q2 = (int *)malloc(sizeof(int));
free(q2);
q2 = 0;    // обнуляем указатель
free(q2); // правильно работает для q2 = 0
```

# Программирование на языке C++

## Лекция 2

Многомерные массивы

Александр Смаль

# Многомерные встроенные массивы

- C++ позволяет определять многомерные массивы:

```
int m2d[2][3] = { {1, 2, 3}, {4, 5, 6} };
for( size_t i = 0; i != 2; ++i ) {
    for( size_t j = 0; j != 3; ++j ) {
        cout << m2d[i][j] << ',';
    }
    cout << endl;
}
```

- Элементы `m2d` располагаются в памяти “по строчкам”.
- Размерность массивов может быть любой, но на практике редко используют массивы размерности  $> 4$ .

```
int m4d[2][3][4][5] = {};
```

## Динамические массивы

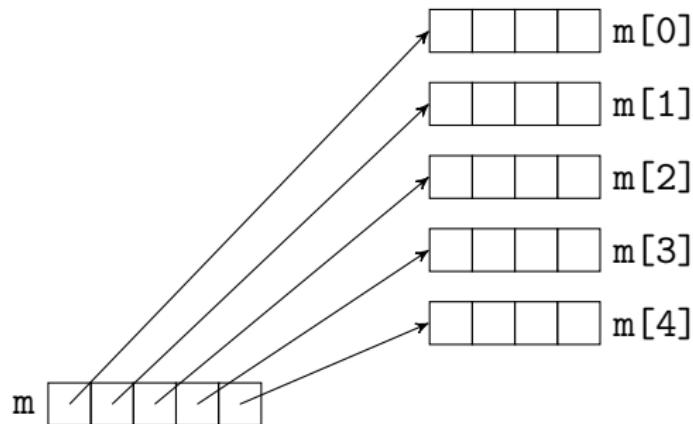
- Для выделения одномерных динамических массивов обычно используется оператор `new []`.

```
int * m1d = new int[100];
```

- Какой тип должен быть у указателя на двумерный динамический массив?
  - Пусть `m` — указатель на двумерный массив типа `int`.
  - Значит `m[i][j]` имеет тип `int` (точнее `int &`).
  - $m[i][j] \Leftrightarrow *(m[i] + j)$ , т.е. тип `m[i]` — `int *`.
  - аналогично, `m[i] \Leftrightarrow *(m + i)`, т.е. тип `m` — `int **`.
- Чему соответствует значение `m[i]`?  
Это адрес строки с номером `i`.
- Чему соответствует значение `m`?  
Это адрес массива с указателями на строки.

## Двумерные массивы

Давайте рассмотрим создание массива  $5 \times 4$ .



```
int ** m = new int * [5];
for (size_t i = 0; i != 5; ++i)
    m[i] = new int [4];
```

## Двумерные массивы

Выделение и освобождение двумерного массива размера  $a \times b$ .

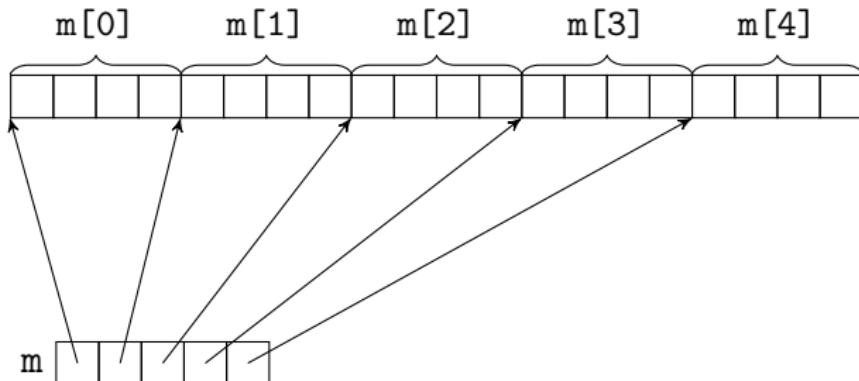
```
int ** create_array2d(size_t a, size_t b) {
    int ** m = new int *[a];
    for (size_t i = 0; i != a; ++i)
        m[i] = new int [b];
    return m;
}

void free_array2d(int ** m, size_t a, size_t b) {
    for (size_t i = 0; i != a; ++i)
        delete [] m[i];
    delete [] m;
}
```

При создании массива оператор `new` вызывается  $(a + 1)$  раз.

## Двумерные массивы: эффективная схема

Рассмотрим эффективное создание массива  $5 \times 4$ .



```
int ** m = new int * [5];
m[0] = new int[5 * 4];
for (size_t i = 1; i != 5; ++i)
    m[i] = m[i - 1] + 4;
```

## Двумерные массивы: эффективная схема

Эффективное выделение и освобождение двумерного массива размера  $a \times b$ .

```
int ** create_array2d(size_t a, size_t b) {
    int ** m = new int *[a];
    m[0] = new int[a * b];
    for (size_t i = 1; i != a; ++i)
        m[i] = m[i - 1] + b;
    return m;
}

void free_array2d(int ** m, size_t a, size_t b) {
    delete [] m[0];
    delete [] m;
}
```

При создании массива оператор `new` вызывается 2 раза.

# Программирование на языке C++

## Лекция 2

Строки и ввод-вывод

Александр Смаль

# Строковые литералы

- Строки — это массивы символов типа `char`, заканчивающиеся нулевым символом.

```
// массив 'H', 'e', 'l', 'l', 'o', '\0'  
char s[] = "Hello";
```

- Строки могут содержать управляемые последовательности:
  1. `\n` — перевод строки,
  2. `\t` — символ табуляции,
  3. `\\` — символ '`\`',
  4. `\"` — символ '`"`',
  5. `\0` — нулевой символ.

```
cout << "List:\n\t- C,\n\t- C++. \n";
```

## Работа со строками в стиле С

- Библиотека `cstring` предлагает множество функций для работы со строками (`char *`).

```
char s1[100] = "Hello";
cout << strlen(s1) << endl; // 5

char s2[] = ", world!";
strcat(s1, s2);

char s3[6] = {72, 101, 108, 108, 111};
if (strcmp(s1, s3) == 0)
    cout << "s1 == s3" << endl;
```

- Работа со строками в стиле С предполагает кропотливую работу с ручным выделением памяти.

## Работа со строками в стиле C++

Библиотека `string` предлагает обёртку над строками, которая позволяет упростить все операции со строками.

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

int main() {
    string s1 = "Hello";
    cout << s1.size() << endl; // 5

    string s2 = ", world!";
    s1 = s1 + s2;

    if (s1 == s2)
        cout << "s1 == s2" << endl;
    return 0;
}
```

## Ввод-вывод в стиле C

- Библиотека `cstdio` предлагает функции для работы со стандартным вводом-выводом.
- Для вывода используется функция `printf`:

```
#include <cstdio>

int main() {
    int h = 20, m = 14;
    printf("Time: %d:%d\n", h, m);
    printf("It's %.2f hours to midnight\n",
           ((24 - h) * 60.0 - m) / 60);
    return 0;
}
```

## Ввод-вывод в стиле C

- Библиотека `cstdio` предлагает функции для работы со стандартным вводом-выводом.
- Для ввода используется функция `scanf`:

```
#include <cstdio>

int main() {
    int a = 0, b = 0;
    printf("Enter a and b: ");
    scanf("%d %d", &a, &b);
    printf("a + b = %d\n", (a + b));
    return 0;
}
```

- Ввод-вывод в стиле C достаточно сложен и небезопасен (типы аргументов не проверяются).

## Ввод-вывод в стиле C++

- В C++ ввод-вывод реализуется через библиотеку `iostream`.

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

int main() {
    string name;
    cout << "Enter your name: ";
    cin >> name; // считывается слово
    cout << "Hi, " << name << endl;

    return 0;
}
```

- Реализация ввода-вывода в стиле C++ типобезопасна.

# Работа с файлами в стиле C++

- Библиотека `fstream` обеспечивает работу с файлами.

```
#include <string>
#include <fstream>
using namespace std;

int main() {
    string name;
    ifstream input("input.txt");
    input >> name;

    ofstream output("output.txt");
    output << "Hi, " << name << endl;
    return 0;
}
```

- Файлы закроются при выходе из функции.

# Программирование на языке C++

## Лекция 3

### Структуры

Александр Смаль

## Зачем группировать данные?

Какая должна быть сигнатура у функции, которая вычисляет длину отрезка на плоскости?

```
double length(double x1, double y1,  
              double x2, double y2);
```

А сигнтура функции, проверяющей пересечение отрезков?

```
bool intersects(double x11, double y11,  
                double x12, double y12,  
                double x21, double y21,  
                double x22, double y22,  
                double * xi, double * yi);
```

Координаты точек являются логически связанными данными, которые всегда передаются вместе.

Аналогично связаны координаты точек отрезка.

# Структуры

Структуры — это способ синтаксически (и физически) сгруппировать логически связанные данные.

```
struct Point {  
    double x;  
    double y;  
};  
  
struct Segment {  
    Point p1;  
    Point p2;  
};  
  
double length(Segment s);  
  
bool intersects(Segment s1,  
                Segment s2, Point * p);
```

## Работа со структурами

Доступ к полям структуры осуществляется через оператор `'.'`:

```
#include <cmath>

double length(Segment s) {
    double dx = s.p1.x - s.p2.x;
    double dy = s.p1.y - s.p2.y;
    return sqrt(dx * dx + dy * dy);
}
```

Для указателей на структуры используется оператор `'->'`.

```
double length(Segment * s) {
    double dx = s->p1.x - s->p2.x;
    double dy = s->p1.y - s->p2.y;
    return sqrt(dx * dx + dy * dy);
}
```

## Инициализация структур

Поля структур можно инициализировать подобно массивам:

```
Point p1 = { 0.4, 1.4 };
Point p2 = { 1.2, 6.3 };
Segment s = { p1, p2 };
```

Структуры могут хранить переменные разных типов.

```
struct IntArray2D {
    size_t a;
    size_t b;
    int ** data;
};
```

```
IntArray2D a = {n, m, create_array2d(n, m)};
```

# Программирование на языке C++

## Лекция 3

Методы

Александр Смаль

## Методы

Метод — это функция, определённая внутри структуры.

```
struct Segment {
    Point p1;
    Point p2;
    double length() {
        double dx = p1.x - p2.x;
        double dy = p1.y - p2.y;
        return sqrt(dx * dx + dy * dy);
    }
};

int main() {
    Segment s = { { 0.4, 1.4 }, { 1.2, 6.3 } };
    cout << s.length() << endl;
    return 0;
}
```

## Методы

Методы реализованы как функции с неявным параметром `this`, который указывает на текущий экземпляр структуры.

```
struct Point
{
    double x;
    double y;

    void shift(/* Point * this, */
              double x, double y) {
        this->x += x;
        this->y += y;
    }
};
```

## Методы: объявление и определение

Методы можно разделять на объявление и определение:

```
struct Point
{
    double x;
    double y;

    void shift(double x, double y);
};
```

```
void Point::shift(double x, double y)
{
    this->x += x;
    this->y += y;
}
```

# Абстракция и инкапсуляция

Использование методов позволяет объединить данные и функции для работы с ними.

```
struct IntArray2D {
    int & get(size_t i, size_t j) {
        return data[i * b + j];
    }
    size_t a;
    size_t b;
    int * data;
};
```

```
IntArray2D m = foo();
for (size_t i = 0; i != m.a; ++i)
    for (size_t j = 0; j != m.b; ++j)
        if (m.get(i, j) < 0) m.get(i, j) = 0;
```

# Программирование на языке C++

## Лекция 3

Конструкторы и деструкторы

Александр Смаль

# Конструкторы

Конструкторы — это методы для инициализации структур.

```
struct Point {  
    Point() {  
        x = y = 0;  
    }  
    Point(double x, double y) {  
        this->x = x;  
        this->y = y;  
    }  
    double x;  
    double y;  
};
```

```
Point p1;  
Point p2(3,7);
```

## Список инициализации

Список инициализации позволяет проинициализировать поля до входа в конструктор.

```
struct Point {
    Point() : x(0), y(0)
    {}
    Point(double x, double y) : x(x), y(y)
    {}

    double x;
    double y;
};
```

Инициализации полей в списке инициализации происходит в порядке объявления полей в структуре.

## Значения по умолчанию

- Функции могут иметь значения параметров *по умолчанию*.
- Значения параметров по умолчанию нужно указывать в *объявлении функции*.

```
struct Point {  
    Point(double x = 0, double y = 0)  
        : x(x), y(y)  
    {}  
    double x;  
    double y;  
};
```

```
Point p1;  
Point p2(2);  
Point p3(3,4);
```

## Конструкторы от одного параметра

Конструкторы от одного параметра задают *неявное* пользовательское преобразование:

```
struct Segment {  
    Segment() {}  
    Segment(double length)  
        : p2(length, 0)  
    {}  
    Point p1;  
    Point p2;  
};
```

```
Segment s1;  
Segment s2(10);  
Segment s3 = 20;
```

## Конструкторы от одного параметра

Для того, чтобы запретить *неявное* пользовательское преобразование, используется ключевое слово `explicit`.

```
struct Segment {  
    Segment() {}  
    explicit Segment(double length)  
        : p2(length, 0)  
    {}  
    Point p1;  
    Point p2;  
};
```

```
Segment s1;  
Segment s2(10);  
Segment s3 = 20; // error
```

## Конструкторы от одного параметра

Неявное пользовательское преобразование, задаётся также конструкторами, которые могут принимать один параметр.

```
struct Point {  
    explicit Point(double x = 0, double y = 0)  
        : x(x), y(y)  
    {}  
    double x;  
    double y;  
};
```

```
Point p1;  
Point p2(2);  
Point p3(3,4);  
Point p4 = 5; // error
```

## Конструктор по умолчанию

Если у структуры нет конструкторов, то конструктор без параметров, *конструктор по умолчанию*, генерируется компилятором.

```
struct Segment {  
    Segment(Point p1, Point p2)  
        : p1(p1), p2(p2)  
    {}  
    Point p1;  
    Point p2;  
};
```

```
Segment s1; // error  
Segment s2(Point(), Point(2,1));
```

# Особенности синтаксиса C++

*“Если что-то похоже на объявление функции, то это и есть объявление функции.”*

```
struct Point {  
    explicit Point(double x = 0, double y = 0)  
        : x(x), y(y) {}  
    double x;  
    double y;  
};
```

```
Point p1;      // определение переменной  
Point p2();   // объявление функции  
  
double k = 5.1;  
Point p3(int(k)); // объявление функции  
Point p4((int)k); // определение переменной
```

## Деструктор

Деструктор — это метод, который вызывается при удалении структуры, генерируется компилятором.

```
struct IntArray {
    explicit IntArray(size_t size)
        : size(size)
        , data(new int[size])
    {}

    ~IntArray() {
        delete [] data;
    }

    size_t size;
    int * data;
};
```

## Время жизни

*Время жизни* — это временной интервал между вызовами конструктора и деструктора.

```
void foo()
{
    IntArray a1(10); // создание a1
    IntArray a2(20); // создание a2
    for (size_t i = 0; i != a1.size; ++i) {
        IntArray a3(30); // создание a3
        ...
    } // удаление a3
} // удаление a2, потом a1
```

Деструкторы переменных на стеке вызываются в обратном порядке (по отношению к порядку вызова конструкторов).

# Программирование на языке C++

## Лекция 3

Объекты и классы

Александр Смаль

# Объекты и классы

- Структуру с методами, конструкторами и деструктором называют *классом*.
- Экземпляр (значение) класса называется *объектом*.

```
struct IntArray {  
    explicit IntArray(size_t size);  
    ~IntArray();  
    int & get(size_t i);  
  
    size_t size;  
    int * data;  
};
```

```
IntArray a(10);  
IntArray b = {20, new int[20]}; // ошибка
```

# Объекты в динамической памяти

## Создание

Для создания объекта в динамической памяти используется оператор `new`, он отвечает за вызов конструктора.

```
struct IntArray {  
    explicit IntArray(size_t size);  
    ~IntArray();  
  
    size_t size;  
    int * data;  
};
```

```
// выделение памяти и создание объекта  
IntArray * pa = new IntArray(10);  
// только выделение памяти  
IntArray * pb =  
    (IntArray *)malloc(sizeof(IntArray));
```

# Объекты в динамической памяти

## Удаление

При вызове оператора `delete` вызывается деструктор объекта.

```
// выделение памяти и создание объекта
IntArray * pa = new IntArray(10);

// вызов деструктора и освобождение памяти
delete pa;
```

Операторы `new []` и `delete []` работают аналогично

```
// выделение памяти и создание 10 объектов
// (вызывается конструктор по умолчанию)
IntArray * pa = new IntArray[10];

// вызов деструкторов и освобождение памяти
delete [] pa;
```

## Placement new

```
// выделение памяти
void * p = myalloc(sizeof(IntArray));

// создание объекта по адресу p
IntArray * a = new (p) IntArray(10);

// явный вызов деструктора
a->~IntArray();

// освобождение памяти
myfree(p);
```

Проблемы с выравниванием:

```
char b[sizeof(IntArray)];
new (b) IntArray(20); // потенциальная проблема
```

# Программирование на языке C++

## Лекция 3

Модификаторы доступа

Александр Смаль

# Модификаторы доступа

Модификаторы доступа позволяют ограничивать доступ к методам и полям класса.

```
struct IntArray {
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_; }

    int & get(size_t i) { return data_[i]; }
    size_t size() { return size_; }

private:
    size_t size_;
    int * data_;
};
```

## Ключевое слово `class`

Ключевое слово `struct` можно заменить на `class`, тогда поля и методы по умолчанию будут `private`.

```
class IntArray {
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_; }

    int & get(size_t i) { return data_[i]; }
    size_t size() { return size_; }

private:
    size_t size_;
    int * data_;
};
```

## Инварианты класса

- Выделение *публичного интерфейса* позволяет поддерживать *инварианты класса* (сохранять данные объекта в согласованном состоянии).

```
struct IntArray {  
    ...  
    size_t size_;  
    int * data_; // массив размера size_  
};
```

- Для сохранения инвариантов класса:
  - все поля должны быть закрытыми,
  - публичные методы должны сохранять инварианты класса.
- Закрытие полей класса позволяет абстрагироваться от способа хранения данных объекта.

# Публичный интерфейс

```
struct IntArray {
    ...
    void resize(size_t nsize) {
        int * ndata = new int[nsize];
        size_t n = nsize > size_ ? size_ : nsize;
        for (size_t i = 0; i != n; ++i)
            ndata[i] = data_[i];
        delete [] data_;
        data_ = ndata;
        size_ = nsize;
    }
private:
    size_t size_;
    int * data_;
};
```

## Абстракция

```
struct IntArray {
public:
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size])
    {}
    ~IntArray() { delete [] data_; }

    int & get(size_t i) { return data_[i]; }
    size_t size() { return size_; }

private:
    size_t size_;
    int * data_;
};
```

# Абстракция

```
struct IntArray {
public:
    explicit IntArray(size_t size)
        : data_(new int[size + 1])
    {
        data_[0] = size;
    }
    ~IntArray() { delete [] data_; }

    int & get(size_t i) { return data_[i + 1]; }
    size_t size() { return data_[0]; }

private:
    int * data_;
};
```

# Программирование на языке C++

## Лекция 3

Константность

Александр Смаль

# Определение констант

- Ключевое слово `const` позволяет определять типизированные константы.

```
double const pi = 3.1415926535;
int const day_seconds = 24 * 60 * 60;
// массив констант
int const days[12] = {31, 28, 31,
                      30, 31, 30,
                      31, 31, 30,
                      31, 30, 31};
```

- Попытка изменить константные данные приводит к неопределённому поведению.

```
int * may = (int *) &days[4];
*may = 30;
```

## Указатели и const

В C++ можно определить как константный указатель, так и указатель на константу:

```
int a = 10;
const int * p1 = &a; // указатель на константу
int const * p2 = &a; // указатель на константу
*p1 = 20; // ошибка
p2 = 0; // OK

int * const p3 = &a; // константный указатель
*p3 = 30; // OK
p3 = 0; // ошибка

// константный указатель на константу
int const * const p4 = &a;
*p4 = 30; // ошибка
p4 = 0; // ошибка
```

## Указатели и const

Можно использовать следующее правило:

“слово `const` делает неизменяемым тип слева от него”.

```
int a = 10;
int * p = &a;

// указатель на указатель на const int
int const ** p1 = &p;

// указатель на константный указатель на int
int * const * p2 = &p;

// константный указатель на указатель на int
int ** const p3 = &p;
```

## Ссылки и const

- Ссылка сама по себе является неизменяемой.

```
int a = 10;
int & const b = a; // ошибка
int const & c = a; // ссылка на константу
```

- Использование константных ссылок позволяет избежать копирования объектов при передаче в функцию.

```
Point midpoint(Segment const & s);
```

- По константной ссылке можно передавать rvalue.

```
Point p = midpoint(Segment(Point(0,0),
                           Point(1,1)));
```

## Константные методы

- Методы классов могут быть объявлены как `const`.

```
struct IntArray {  
    size_t size() const;  
};
```

- Такие методы не могут менять поля объекта (тип `this` — указатель на `const`).
- У константных объектов (через указатель или ссылку на константу) можно вызывать только константные методы:

```
IntArray const * p = foo();  
p->resize(); // ошибка
```

- Внутри константных методов можно вызывать только константные методы.

## Две версии одного метода

- Слово `const` является частью сигнатуры метода.

```
size_t IntArray::size() const {return size_ ;}
```

- Можно определить две версии одного метода:

```
struct IntArray {
    int get(size_t i) const {
        return data_[i];
    }
    int & get(size_t i) {
        return data_[i];
    }
private:
    size_t size_;
    int * data_;
};
```

## Синтаксическая и логическая константность

- Синтаксическая константность: константные методы не могут менять поля (обеспечивается компилятором).
- Логическая константность — нельзя менять те данные, которые определяют состояние объекта.

```
struct IntArray {
    void foo() const {
        // нарушение логической константности
        data_[10] = 1;
    }
private:
    size_t size_;
    int * data_;
};
```

## Ключевое слово `mutable`

Ключевое слово `mutable` позволяет определять поля, которые можно изменять внутри константных методов:

```
struct IntArray {
    size_t size() const {
        ++counter_;
        return size_;
    }

private:
    size_t size_;
    int * data_;

    mutable size_t counter_;
};
```

# Программирование на языке C++

## Лекция 3

Конструктор копирования и оператор  
присваивания

Александр Смаль

## Копирование объектов

```
struct IntArray {  
    ...  
private:  
    size_t size_;  
    int * data_;  
};  
  
int main() {  
    IntArray a1(10);  
    IntArray a2(20);  
    IntArray a3 = a1; // копирование  
    a2 = a1; // присваивание  
  
    return 0;  
}
```

## Конструктор копирования

Если не определить конструктор копирования, то он сгенерируется компилятором.

```
struct IntArray {
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_])
    {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

## Оператор присваивания

Если не определить оператор присваивания, то он тоже сгенерируется компилятором.

```
struct IntArray {
    IntArray & operator=(IntArray const& a)
    {
        if (this != &a) {
            delete [] data_;
            size_ = a.size_;
            data_ = new int[size_];
            for (size_t i = 0; i != size_; ++i)
                data_[i] = a.data_[i];
        }
        return *this;
    }
    ...
};
```

## Метод swap

```
struct IntArray {
    void swap(IntArray & a) {
        size_t const t1 = size_;
        size_ = a.size_;
        a.size_ = t1;

        int * const t2 = data_;
        data_ = a.data_;
        a.data_ = t2;
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

## Метод swap

Можно использовать функцию `std::swap` и файла [algorithm](#).

```
#include <algorithm>

struct IntArray {
    void swap(IntArray & a) {
        std::swap(size_, a.size_);
        std::swap(data_, a.data_);
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

## Реализация оператора = при помощи swap

```
struct IntArray {
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    IntArray & operator=(IntArray const& a) {
        if (this != &a)
            IntArray(a).swap(*this);
        return *this;
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

## Запрет копирования объектов

Для того, чтобы запретить копирование, нужно объявить конструктор копирования и оператор присваивания как `private` и не определять их.

```
struct IntArray {  
    ...  
private:  
    IntArray(IntArray const& a);  
    IntArray & operator=(IntArray const& a);  
  
    size_t size_;  
    int * data_;  
};
```

## Методы, генерируемые компилятором

Компилятор генерирует четыре метода:

1. конструктор по умолчанию,
2. конструктор копирования,
3. оператор присваивания,
4. деструктор.

Если потребовалось переопределить конструктор копирования, оператор присваивания или деструктор, то нужно переопределить и остальные методы из этого списка.

# Программирование на языке C++

## Лекция 3

Класс массива

Александр Смаль

## Поля и конструкторы

```
struct IntArray {
    explicit IntArray(size_t size)
        : size_(size), data_(new int[size]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = 0;
    }
    IntArray(IntArray const& a)
        : size_(a.size_), data_(new int[size_]) {
        for (size_t i = 0; i != size_; ++i)
            data_[i] = a.data_[i];
    }
    ...
private:
    size_t size_;
    int * data_;
};
```

## Деструктор, оператор присваивания и swap

```
~IntArray() {
    delete [] data_;
}

IntArray & operator=(IntArray const& a) {
    if (this != &a)
        IntArray(a).swap(*this);
    return *this;
}

void swap(IntArray & a) {
    std::swap(size_, a.size_);
    std::swap(data_, a.data_);
}
```

## Методы

```
size_t size() const { return size_; }

int     get(size_t i) const {
    return data_[i];
}
int &   get(size_t i)          {
    return data_[i];
}

void resize(size_t nsize) {
    IntArray t(nsize);
    size_t n = nsize > size_ ? size_ : nsize;
    for (size_t i = 0; i != n; ++i)
        t.data_[i] = data_[i];
    swap(t);
}
```

# Программирование на языке C++

## Лекция 4

Наследование

Александр Смаль

## Наследование

Наследование — это механизм, позволяющий создавать производные классы, расширяя уже существующие.

```
struct Person {
    string name() const { return name_; }
    int     age()  const { return age_; }

private:
    string name_;
    int   age_;

};

struct Student : Person {
    string university() const { return uni_; }

private:
    string uni_;

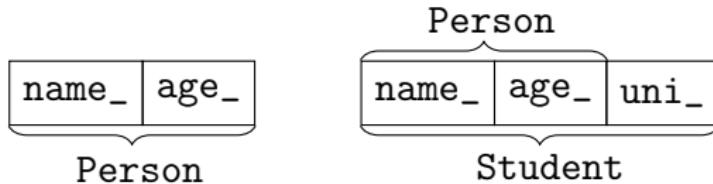
};
```

## Класс-наследник

У объектов класса-наследника можно вызывать публичные методы родительского класса.

```
Student s;  
cout << s.name() << endl  
    << s.age() << endl  
    << s.university() << endl;
```

Внутри объекта класса-наследника хранится экземпляр родительского класса.



## Создание/удаление объекта производного класса

При создании объекта производного класса сначала вызывается конструктор родительского класса.

```
struct Person {
    Person(string name, int age)
        : name_(name), age_(age)
    {}
    ...
};

struct Student : Person {
    Student(string name, int age, string uni)
        : Person(name, age), uni_(uni)
    {}
    ...
};
```

После деструктора Student вызывается деструктор Person.

## Приведения

Для производных классов определены следующие приведения:

```
Student s("Alex", 21, "Oxford");
Person & l = s; // Student & -> Person &
Person * r = &s; // Student * -> Person *
```

Поэтому объекты класса-наследника могут присваиваться объектам родительского класса:

```
Student s("Alex", 21, "Oxford");
Person p = s; // Person("Alex", 21);
```

При этом копируются только поля класса-родителя (срезка).  
(Т.е. в данном случае вызывается конструктор копирования  
Person(Person const& p), который не знает про uni\_.)

## Модификатор доступа protected

- Класс-наследник не имеет доступа к private-членам родительского класса.
- Для определения закрытых членов класса доступных наследникам используется модификатор `protected`.

```
struct Person {  
    ...  
protected:  
    string name_;  
    int age_;  
};  
  
struct Student : Person {  
    ... // можно менять поля name_ и age_  
};
```

# Программирование на языке C++

## Лекция 4

Перегрузка

Александр Смаль

# Перегрузка функций

В отличие от С в С++ можно определить несколько функций с одинаковым именем, но разными параметрами.

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

int     square(int     i) { return i * i; }
```

При вызове функции по имени будет произведен поиск наиболее подходящей функции:

```
int      a = square(4);           // square(int)
double   b = square(3.14);        // square(double)
double   c = square(5);          // square(int)
int      d = square(b);          // square(double)
float    e = square(2.71f);       // square(double)
```

## Перегрузка методов

```
struct Vector2D {  
    Vector2D(double x, double y) : x(x), y(y) {}  
  
    Vector2D mult(double d) const  
    { return Vector2D(x * d, y * d); }  
  
    double mult(Vector2D const& p) const  
    { return x * p.x + y * p.y; }  
  
    double x, y;  
};
```

```
Vector2D p(1, 2);  
Vector2D q = p.mult(10); // (10, 20)  
double r = p.mult(q); // 50
```

## Перегрузка при наследовании

```
struct File {  
    void write(char const * s);  
    ...  
};  
  
struct FormattedFile : File {  
    void write(int i);  
    void write(double d);  
    using File::write;  
    ...  
};
```

```
FormattedFile f;  
f.write(4);  
f.write("Hello");
```

# Правила перегрузки

1. Если есть точное совпадение, то используется оно.
2. Если нет функции, которая могла бы подойти с учётом преобразований, выдаётся ошибка.
3. Есть функции, подходящие с учётом преобразований:

## 3.1 Расширение типов.

`char, signed char, short → int`

`unsigned char, unsigned short → int/unsigned int`  
`float → double`

## 3.2 Стандартные преобразования (числа, указатели).

## 3.3 Пользовательские преобразования.

В случае нескольких параметров нужно, чтобы выбранная функция была *строго лучше* остальных.

**NB:** перегрузка выполняется на этапе компиляции.

# Программирование на языке C++

## Лекция 4

Виртуальные методы

Александр Смаль

## Переопределение методов (overriding)

```
struct Person {  
    string name() const { return name_; }  
    ...  
};  
struct Professor : Person {  
    string name() const {  
        return "Prof. " + Person::name();  
    }  
    ...  
};
```

```
Professor pr("Stroustrup");  
cout << pr.name() << endl; // Prof. Stroustrup  
Person * p = &pr;  
cout << p->name() << endl; // Stroustrup
```

## Виртуальные методы

```
struct Person {
    virtual string name() const { return name_; }
    ...
};

struct Professor : Person {
    string name() const {
        return "Prof. " + Person::name();
    }
    ...
};
```

```
Professor pr("Stroustrup");
cout << pr.name() << endl; // Prof. Stroustrup
Person * p = &pr;
cout << p->name() << endl; // Prof. Stroustrup
```

## Чистые виртуальные (абстрактные) методы

```
struct Person {
    virtual string occupation() const = 0;
    ...
};

struct Student : Person {
    string occupation() const {return "student";}
    ...
};

struct Professor : Person {
    string occupation() const {return "professor";}
    ...
};
```

```
Person * p = next_person();
cout << p->occupation();
```

# Виртуальный деструктор

К чему приведёт такой код?

```
struct Person {
    ...
};

struct Student : Person {
    ...
private:
    string uni_;
};

int main() {
    Person * p = new Student("Alex", 21, "Oxford");
    ...
    delete p;
}
```

# Виртуальный деструктор

Правильная реализация:

```
struct Person {
    ...
    virtual ~Person() {}
};

struct Student : Person {
    ...
private:
    string uni_;
};

int main() {
    Person * p = new Student("Alex", 21, "Oxford");
    ...
    delete p;
}
```

# Полиморфизм

## Полиморфизм

Возможность единообразно обрабатывать разные типы данных.

## Перегрузка функций

Выбор функции происходит в момент компиляции на основе типов аргументов функции, *статический полиморфизм*.

## Виртуальные методы

Выбор метода происходит в момент выполнения на основе типа объекта, у которого вызывается виртуальный метод, *динамический полиморфизм*.

# Программирование на языке C++

## Лекция 4

Таблица виртуальных методов

Александр Смаль

## Таблица виртуальных методов

- Динамический полиморфизм реализуется при помощи таблиц виртуальных методов.
- Таблица заводится для каждого *полиморфного* класса.
- Объекты полиморфных классов содержат указатель на таблицу виртуальных методов соответствующего класса.



- Вызов виртуального метода — это вызов метода по адресу из таблицы (в коде сохраняется номер метода в таблице).

```
p->occupation(); // p->vptr[1]();
```

## Таблица виртуальных методов

```
struct Person {
    virtual ~Person() {}
    string name() const {return name_;}
    virtual string occupation() const = 0;
    ...
};

struct Student : Person {
    string occupation() const {return "student";}
    virtual int group() const {return group_;}
    ...
};
```

Person

0	~Person	0xab22
1	occupation	0x0000

Student

0	~Student	0xab46
1	occupation	0xab68
2	group	0xab8a

# Построение таблицы виртуальных методов

```
struct Person {
    virtual ~Person() {}
    virtual string occupation() = 0;
    ...
};

struct Teacher : Person {
    string occupation() {...}
    virtual string course() {...}
    ...
};

struct Professor : Teacher {
    string occupation() {...}
    virtual string thesis() {...}
    ...
};
```

Person

0	~Person	0xab20
1	occupation	0x0000

Teacher

0	~Teacher	0xab48
1	occupation	0xab60
2	course	0xab84

Professor

0	~Professor	0xaba8
1	occupation	0xabbb4
2	course	0xab84
3	thesis	0xabcc8

## Виртуальные методы в конструкторе и деструкторе

```
struct Person {
    virtual string name() const {return name_;}
    ...
};

struct Teacher : Person {
    Teacher(string const& nm) : Person(nm)
    { cout << name(); }
    ...
};

struct Professor : Teacher {
    string name() const {return "Prof. "+name_;}
    ...
};
```

```
Professor p("Stroustrup"); // "Stroustrup"
```

# Программирование на языке C++

## Лекция 4

Объектно-ориентированное программирование

Александр Смаль

# Ещё раз об ООП

*Объектно-ориентированное программирование — концепция программирования, основанная на понятиях объектов и классов.*

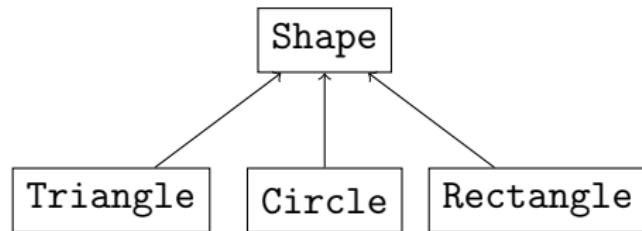
**Основные принципы:**

- инкапсуляция,
- наследование,
- полиморфизм,
- абстракция.

Подробнее о принципах проектирования ООП-программ можно узнать по ключевым слову „шаблоны проектирования”.

# Как правильно построить иерархию?

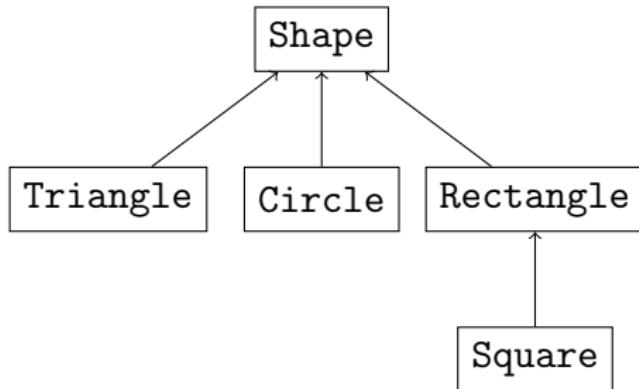
Иерархия геометрических фигур:



Куда добавить класс Square?

# Как правильно построить иерархию?

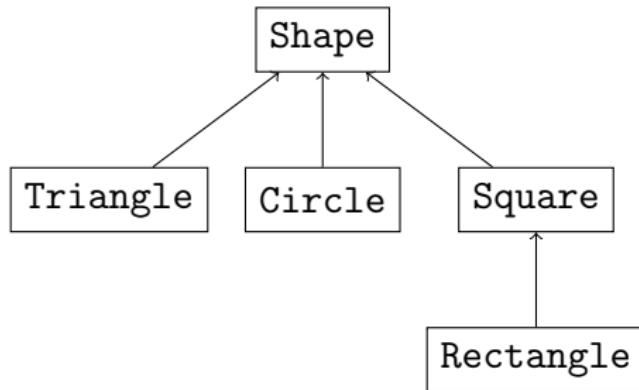
Квадрат — это прямоугольник, у которого все стороны равны.



```
void double_width(Rectangle & r) {  
    r.set_width(r.width() * 2);  
}
```

# Как правильно построить иерархию?

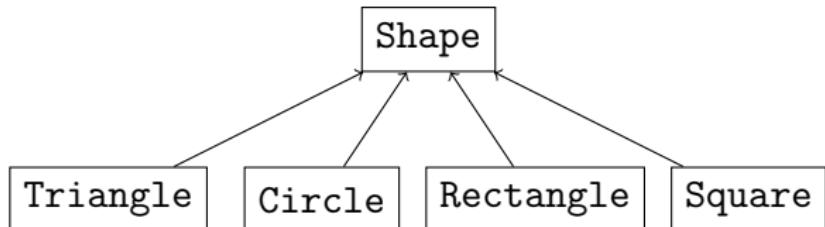
Прямоугольник задаётся двумя сторонами, а квадрат — одной.



```
double area(Square const& s) {  
    return s.width() * s.width();  
}
```

# Как правильно построить иерархию?

Правильное решение — сделать эти классы независимыми:



## Агрегирование vs наследование

- Агрегирование — это включение объекта одного класса в качестве поля в другой.
- Наследование устанавливает более сильные связи между классами, нежели агрегирование:
  - приведение между объектами,
  - доступ к `protected` членам.
- Если наследование можно заменить легко на агрегирование, то это нужно сделать.

## Примеры некорректного наследования

- Класс `Circle` унаследовать от класса `Point`.
- Класс `LinearSystem` унаследовать от класса `Matrix`.

# Принцип подстановки Барбары Лисков

## Liskov Substitution Principle (LSP)

*Функции, работающие с базовым классом, должны иметь возможность работать с подклассами не зная об этом.*

Этот принцип является важнейшим критерием при построении иерархий наследования.

## Другие формулировки

- Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом.
- Подкласс не должен требовать от вызывающего кода больше, чем базовый класс, и не должен предоставлять вызывающему коду меньше, чем базовый класс

# Программирование на языке C++

## Лекция 4

Особенности наследования в C++

Александр Смаль

# Модификаторы при наследовании

При наследовании можно использовать модификаторы доступа:

```
struct A {};
struct B1 : public A {};
struct B2 : private A {};
struct B3 : protected A {};
```

Для классов, объявленных как `struct`, по-умолчанию используется `public`, для объявленных как `class` — `private`.

**Важно:** отношение наследования (в терминах ООП) задаётся только `public`-наследованием.

Использование `private`- и `protected`-наследований целесообразно, если необходимо не только агрегировать другой класс, но и переопределить его виртуальные методы.

## Переопределение private виртуальных методов

```
struct NetworkDevice {
    void send(void * data, size_t size) {
        log("start sending");
        send_impl(data, size);
        log("stop sending");
    }
    ...
private:
    virtual void send_impl(void * data, size_t size)
    {...}
};

struct Router : NetworkDevice {
private:
    void send_impl(void * data, size_t size) {...}
};
```

# Реализация чистых виртуальных методов

Чистые виртуальные методы могут иметь определения:

```
struct NetworkDevice {
    virtual void send(void * data, size_t size) = 0;
    ...
};

void NetworkDevice::send(void * data, size_t size) {
    ...
}

struct Router : NetworkDevice {
private:
    void send(void * data, size_t size) {
        // невиртуальный вызов
        NetworkDevice::send(data, size);
    }
};
```

# Интерфейсы

Интерфейс — это абстрактный класс, у которого отсутствуют поля, а все методы являются чистыми виртуальными.

```
struct IConvertibleToString {
    virtual ~IConvertibleToString() {}
    virtual string toString() const = 0;
};
```

```
struct IClonable {
    virtual ~IClonable() {}
    virtual IClonable * clone() const = 0;
};
```

```
struct Person : IClonable {
    Person * clone() {return new Person(*this);}
};
```

# Множественное наследование

В C++ разрешено множественное наследование.

```
struct Person {};
struct Student : Person {};
struct Worker : Person {};
struct WorkingStudent : Student, Worker {};
```

Стоит избегать *наследования реализаций* более чем от одного класса, вместо этого использовать интерфейсы.

```
struct IWorker {};
struct Worker : Person, IWorker {};
struct Student : Person {};
struct WorkingStudent : Student, IWorker {}
```

Множественное наследование — это отдельная большая тема.

# Программирование на языке C++

## Лекция 5

Перегрузка операторов

Александр Смаль

# Основные операторы

## Арифметические

- Унарные: префиксные `+` `-` `++` `--`, постфиксные `++` `--`
- Бинарные: `+` `-` `*` `/` `%` `+=` `-=` `*=` `/=` `%=`

## Битовые

- Унарные: `~`.
- Бинарные: `&` `|` `^` `&=` `|=` `^=` `>>` `<<`.

## Логические

- Унарные: `!`.
- Бинарные: `&&` `||`.
- Сравнения: `==` `!=` `>` `<` `>=` `<=`

# Другие операторы

1. Оператор присваивания: `=`
2. Специальные:
  - префиксные `*` `&`,
  - постфиксные `->` `->*`,
  - особые `, . ::`
3. Скобки: `[] ()`
4. Оператор приведения (`type`)
5. Тернарный оператор: `x ? y : z`
6. Работа с памятью: `new new[] delete delete[]`

Нельзя перегружать операторы `. ::` и тернарный оператор.

## Перегрузка операторов

```
Vector operator-(Vector const& v) {
    return Vector(-v.x, -v.y);
}

Vector operator+(Vector const& v,
                  Vector const& w) {
    return Vector(v.x + w.x, v.y + w.y);
}

Vector operator*(Vector const& v, double d) {
    return Vector(v.x * d, v.y * d);
}

Vector operator*(double d, Vector const& v) {
    return v * d;
}
```

# Перегрузка операторов внутри классов

**NB:** Обязательно для `(type) [] () -> ->* =`

```
struct Vector {
    Vector operator-() const { return Vector(-x, -y); }
    Vector operator-(Vector const& p) const {
        return Vector(x - p.x, y - p.y);
    }
    Vector & operator*=(double d) {
        x *= d;
        y *= d;
        return *this;
    }
    double operator[](size_t i) const {
        return (i == 0) ? x : y;
    }
    bool operator()(double d) const { ... }
    void operator()(double a, double b) { ... }
    double x, y;
};
```

## Перегрузка инкремента и декремента

```
struct BigNum {  
    BigNum & operator++() { //prefix  
        //increment  
        ...  
        return *this;  
    }  
  
    BigNum operator++(int) { //postfix  
        BigNum tmp(*this);  
        ++(*this);  
        return tmp;  
    }  
    ...  
};
```

## Переопределение операторов ввода-вывода

```
#include <iostream>

struct Vector { ... };

std::istream& operator>>(std::istream & is,
                           Vector & p) {
    is >> p.x >> p.y;
    return is;
}

std::ostream& operator<<(std::ostream &os,
                           Vector const& p) {
    os << p.x << ',' << p.y;
    return os;
}
```

## Умный указатель

Реализует принцип: “Получение ресурса есть инициализация”  
*Resource Acquisition Is Initialization (RAII)*

```
struct SmartPtr {
    Data & operator*() const {return *data_;}
    Data * operator->() const {return data_;}
    Data * get() const {return data_;}
    ...
private:
    Data * data_;
};

bool operator==(SmartPtr const& p1,
                  SmartPtr const& p2) {
    return p1.get() == p2.get();
}
```

## Оператор приведения

```
struct String {
    operator bool() const {
        return size_ != 0;
    }

    operator char const *() const {
        if (*this)
            return data_;
        return "";
    }

private:
    char * data_;
    size_t size_;
};
```

## Операторы с особым порядком вычисления

```
int main() {
    int a = 0;
    int b = 5;
    (a != 0) && (b = b / a);
    (a == 0) || (b = b / a);

    foo() && bar();
    foo() || bar();
    foo(), bar();
}

// no lazy semantics
Tribool operator&&(Tribool const& b1,
                      Tribool const& b2) {
    ...
}
```

# Программирование на языке C++

## Лекция 5

Правила переопределения операторов

Александр Смаль

## Переопределение арифметических и битовых операторов

```
struct String {  
    String( char const * cstr ) { ... }  
  
    String & operator+=(String const& s) {  
        ...  
        return *this;  
    }  
    //String operator+(String const& s2) const {...}  
};  
  
String operator+(String s1, String const& s2) {  
    return s1 += s2;  
}
```

```
String s1("world");  
String s2 = "Hello " + s1;
```

## “Правильное” переопределение операторов сравнения

```
bool operator==(String const& a, String const& b) {
    return ...
}
bool operator!=(String const& a, String const& b) {
    return !(a == b);
}
bool operator<(String const& a, String const& b) {
    return ...
}
bool operator>(String const& a, String const& b) {
    return b < a;
}
bool operator<=(String const& a, String const& b) {
    return !(b < a);
}
bool operator>=(String const& a, String const& b) {
    return !(a < b);
}
```

# О чём стоит помнить

- Стандартная семантика операторов.

```
void operator+(A const & a, A const& b) {}
```

- Приоритет операторов.

```
Vector a, b, c;  
c = a + a ^ b * a; //?????
```

- Хотя бы один из параметров должен быть пользовательским.

```
void operator*(double d, int i) {}
```

# Программирование на языке C++

## Лекция 5

Ключевые слова `static` и `inline`

Александр Смаль

# Глобальные переменные

Объявление глобальной переменной:

```
extern int global;

void f () {
    ++global;
}
```

Определение глобальной переменной:

```
int global = 10;
```

Проблемы глобальных переменных:

- Масштабируемость.
- Побочные эффекты.
- Порядок инициализации.

## Статические глобальные переменные

Статическая глобальная переменная — это глобальная переменная, доступная только в пределах модуля.

Определение:

```
static int global = 10;

void f () {
    ++global;
}
```

Проблемы статических глобальных переменных:

- Масштабируемость.
- Побочные эффекты.

## Статические локальные переменные

Статическая локальная переменная — это глобальная переменная, доступная только в пределах функции.

Время жизни такой переменной — от первого вызова функции `next` до конца программы.

```
int next(int start = 0) {  
    static int k = start;  
    return k++;  
}
```

Проблемы статических локальных переменных:

- Масштабируемость.
- Побочные эффекты.

# Статические функции

Статическая функция, доступная только в пределах модуля.

Файл 1.cpp:

```
static void test() {
    cout << "A\n";
}
```

Файл 2.cpp:

```
static void test() {
    cout << "B\n";
}
```

Статические глобальные переменные и статические функции проходят *внутреннюю линковку*.

## Статические поля класса

Статические поля класса — это глобальные переменные, определённые внутри класса.

Объявление:

```
struct User {  
    ...  
private:  
    static size_t instances_;  
};
```

Определение:

```
size_t User::instances_ = 0;
```

Для доступа к статическим полям не нужен объект.

## Статические методы

Статические методы — это функции, определённые внутри класса и имеющие доступ к закрытым полям и методам.

Объявление:

```
struct User {  
    ...  
    static size_t count() { return instances_; }  
private:  
    static size_t instances_;  
};
```

Для вызова статических методов не нужен объект.

```
cout << User::count();
```

## Ключевое слово `inline`

Советует компилятору встроить данную функцию.

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

- В месте вызова `inline`-функции должно быть известно её определение.
- `inline` функции можно определять в заголовочных файлах.
- Все методы, определённые внутри класса, являются `inline`.
- При линковке из всех версий `inline`-функции (т.е. её код из разных единиц трансляции) выбирается только одна.
- Все определения одной и той же `inline`-функции должны быть идентичными.
- `inline` — это совет компилятору, а не указ.

# Правило одного определения

## Правило одного определения (One Definition Rule, ODR)

- В пределах любой единицы трансляции сущности не могут иметь более одного определения.
- В пределах программы глобальные переменные и не-`inline` функции не могут иметь больше одного определения.
- Классы и `inline` функции могут определяться в более чем одной единице трансляции, но определения обязаны совпадать.

Вопрос: к каким проблемам могут привести разные определения одного класса в разных частях программы?

# Программирование на языке C++

## Лекция 5

Ключевое слово `friend`

Александр Смаль

## Дружественные классы

```
struct String {  
    ...  
    friend struct StringBuffer;  
private:  
    char * data_;  
    size_t len_;  
};  
  
struct StringBuffer {  
    void append(String const& s) {  
        append(s.data_);  
    }  
    void append(char const* s) {...}  
    ...  
};
```

## Дружественные функции

Дружественные функции можно определять прямо внутри описания класса (они становятся [inline](#)).

```
struct String {  
    ...  
    friend std::ostream&  
        operator<<(std::ostream & os,  
                      String const& s)  
    {  
        return os << s.data_;  
    }  
  
private:  
    char * data_;  
    size_t len_;  
};
```

## Дружественные методы

```
struct String;
struct StringBuffer {
    void append(String const& s);
    void append(char const* s) {...}
    ...
};

struct String {
    ...
    friend
        void StringBuffer::append(String const& s);
};

void StringBuffer::append(String const& s) {
    append(s.data_);
}
```

# Отношение дружбы

Отношение дружбы можно охарактеризовать следующими утверждениями:

- Отношение дружбы не симметрично.
- Отношение дружбы не транзитивно.
- Отношение наследования не задаёт отношение дружбы.
- Отношение дружбы сильнее, чем отношение наследования.

## Вывод

Стоит избегать ключевого слова `friend`, так как оно нарушает инкапсуляцию.

# Программирование на языке C++

## Лекция 5

Шаблон проектирования Singleton

Александр Смаль

## Класс Singleton

```
struct Singleton {
    static Singleton & instance() {
        static Singleton s;
        return s;
    }

    Data & data() { return data_; }

private:
    Singleton() {}

    Singleton(Singleton const&);
    Singleton& operator=(Singleton const&);

    Data data_;
};
```

## Использование Singleton-а

```
int main()
{
    // первое обращение
    Singleton & s = Singleton::instance();
    Data d = s.data();

    // аналогично d = s.data();
    d = Singleton::instance().data();
    return 0;
}
```

# Программирование на языке C++

## Лекция 6

Шаблоны классов

Александр Смаль

# Проблема “одинаковых классов”

```
struct ArrayInt {  
    explicit ArrayInt(size_t size)  
        : data_(new int[size])  
        , size_(size) {}  
  
    ~ArrayInt() {delete [] data_;}  
  
    size_t size() const  
    { return size_; }  
  
    int operator[](size_t i) const  
    { return data_[i]; }  
  
    int & operator[](size_t i)  
    { return data_[i]; }  
    ...  
private:  
    int * data_;  
    size_t size_;  
};
```

```
struct ArrayFlt {  
    explicit ArrayFlt(size_t size)  
        : data_(new float[size])  
        , size_(size) {}  
  
    ~ArrayFlt() {delete [] data_;}  
  
    size_t size() const  
    { return size_; }  
  
    float operator[](size_t i) const  
    { return data_[i]; }  
  
    float & operator[](size_t i)  
    { return data_[i]; }  
    ...  
private:  
    float * data_;  
    size_t size_;  
};
```

## Решение в стиле С: макросы

```
#define DEFINE_ARRAY(Name, Type) \
struct Name { \
    explicit Name(size_t size) \
        : data_(new Type[size]) \
        , size_(size) {} \
    ~Name() { delete [] data_; } \
    \
    size_t size() const \
    { return size_; } \
    \
    Type operator[](size_t i) const \
    { return data_[i]; } \
    Type & operator[](size_t i) \
    { return data_[i]; } \
    ... \
private: \
    Type * data_; \
    size_t size_; \
}
```

```
DEFINE_ARRAY(ArrayInt, int); \
DEFINE_ARRAY(ArrayFlt, float); \
 \
int main() \
{ \
    ArrayInt ai(10); \
    ArrayFlt af(20); \
    ... \
    return 0; \
}
```

## Решение в стиле C++: шаблоны классов

```
template <class Type>
struct Array {
    explicit Array(size_t size)
        : data_(new Type[size])
        , size_(size) {}

    ~Array()
    { delete [] data_; }

    size_t size() const
    { return size_; }

    Type operator[](size_t i) const
    { return data_[i]; }

    Type & operator[](size_t i)
    { return data_[i]; }

    ...

    private:
        Type * data_;
        size_t size_;
};
```

```
int main()
{
    Array<int> ai(10);
    Array<float> af(20);

    ...
    return 0;
}
```

# Шаблоны классов с несколькими параметрами

```
template <class Type,
          class SizeT = size_t,
          class CRet = Type>
struct Array {
    explicit Array(SizeT size)
        : data_(new Type[size])
        , size_(size) {}
    ~Array() {delete [] data_;}
    SizeT size() const {return size_;}
    CRet operator[](SizeT i) const
    { return data_[i]; }
    Type & operator[](SizeT i)
    { return data_[i]; }
    ...
private:
    Type * data_;
    SizeT size_;
};
```

```
void foo()
{
    Array<int> ai(10);
    Array<float> af(20);
    Array<Array<int>,
           size_t,
           Array<int> const &>
    da(30);
    ...
}

typedef Array<int> Ints;
typedef Array<Ints, size_t,
             Ints const &> IIInts;

void bar()
{
    IIInts da(30);
}
```

# Программирование на языке C++

## Лекция 6

### Шаблоны функций

Александр Смаль

# Шаблоны функций: возвведение в квадрат

```
// C
int    squarei(int    x)    { return x * x; }
float  squaref(float  x)    { return x * x; }

// C++
int    square(int    x)    { return x * x; }
float  square(float  x)    { return x * x; }

// C++ + OOP
struct INumber {
    virtual INumber * multiply(INumber * x) const = 0;
};

struct Int    : INumber { ... };
struct Float : INumber { ... };

INumber * square(INumber * x) { return x->multiply(x); }

// C++ + templates
template <typename Num>
Num square(Num x) { return x * x; }
```

# Шаблоны функций: сортировка

```
// C
void qsort(void * base, size_t nitems, size_t size, /*function*/);

// C++
void sort(int      * p,      int      * q);
void sort(double   * p,      double   * q);

// C++ + OOP
struct IComparable {
    virtual int compare(IComparable * comp) const = 0;
    virtual ~IComparable() {}
};

void sort(IComparable ** p, IComparable ** q);

// C++ + templates
template <typename Type>
void sort(Type * p, Type * q);
```

**NB:** у шаблонных функций нет параметров по умолчанию.

## Вывод аргументов (deduce)

```
template <typename Num>
Num square(Num n) { return n * n; }

template <typename Type>
void sort(Type * p, Type * q);

template <typename Type>
void sort(Array<Type> & ar);

void foo() {
    int a = square<int>(3);
    int b = square(a) + square(4); // square<int>(..)
    float * m = new float[10];
    sort(m, m + 10);           // sort<float>(m, m + 10)
    sort(m, &a);              // error: sort<float> vs. sort<int>
    Array<double> ad(100);
    sort(ad);                 // sort<double>(ad)
}
```

# Шаблоны методов

```
template <class Type>
struct Array {
    template<class Other>
    Array( Array<Other> const& other )
        : data_(new Type[other.size()])
        , size_(other.size()) {
        for(size_t i = 0; i != size_; ++i)
            data_[i] = other[i];
    }

    template<class Other>
    Array & operator=(Array<Other> const& other);
    ...
};

template<class Type>
template<class Other>
Array<Type> & Array<Type>::operator=(Array<Other> const& other)
{ ... return *this; }
```

# Программирование на языке C++

## Лекция 6

Специализация шаблонов

Александр Смаль

## Полная специализация шаблонов: классы

```
template<class T>
struct Array {
    ...
    T *    data_;
};

template<>
struct Array<bool> {
    static int const INTBITS = 8 * sizeof(int);
    explicit Array(size_t size)
        : size_(size)
        , data_(new int[size_ / INTBITS + 1])
    {}
    bool operator[](size_t i) const {
        return data_[i / INTBITS] & (1 << (i % INTBITS));
    }
private:
    size_t    size_;
    int *    data_;
};
```

## Полная специализация шаблонов: функции

```
template<class T>
void swap(T & a, T & b)
{
    T tmp(a);
    a = b;
    b = tmp;
}

template<>
void swap<Database>(Database & a, Database & b)
{
    a.swap(b);
}

template<class T>
void swap(Array<T> & a, Array<T> & b)
{
    a.swap(b);
}
```

# Специализация шаблонов функций и перегрузка

```
template<class T>
void foo(T a, T b) {
    cout << "same types" << endl;
}

template<class T, class V>
void foo(T a, V b) {
    cout << "different types" << endl;
}

template<>
void foo<int, int>(int a, int b) {
    cout << "both parameters are int" << endl;
}

int main() {
    foo(3, 4);
    return 0;
}
```

# Частичная специализация шаблонов

```
template<class T>
struct Array {
    T & operator[](size_t i) { return data_[i]; }
    ...
};

template<class T>
struct Array<T *> {
    explicit Array(size_t size)
        : size_(size)
        , data_(new T *[size_])
    {}

    T & operator[](size_t i) { return *data_[i]; }

private:
    size_t    size_;
    T **     data_;
};
```

# Программирование на языке C++

## Лекция 6

Ещё о шаблонах

Александр Смаль

# Нетиповые шаблонные параметры

Параметрами шаблона могут быть типы, целочисленные значения, указатели/ссылки на значения с внешней линковкой и шаблоны.

```
template<class T, size_t N, size_t M>
struct Matrix {
    ...
    T & operator()(size_t i, size_t j)
    { return data_[M * j + i]; }

private:
    T data_[N * M];
};

template<class T, size_t N, size_t M, size_t K>
Matrix<T, N, K> operator*(Matrix<T, N, M> const& a,
                           Matrix<T, M, K> const& b);

// log - это глобальная переменная
template<ofstream & log>
struct FileLogger { ... };
```

# Шаблонные параметры — шаблоны

```
// int -> string
string toString( int i );

// работает только с Array<>
Array<string> toStrings( Array<int> const& ar ) {
    Array<string> result(ar.size());
    for (size_t i = 0; i != ar.size(); ++i)
        result.get(i) = toString(ar.get(i));
    return result;
}

// от контейнера требуется: конструктор от size, методы size() и get()
template<template <class> class Container>
Container<string> toStrings( Container<int> const& c ) {
    Container<string> result(c.size());
    for (size_t i = 0; i != c.size(); ++i)
        result.get(i) = toString(c.get(i));
    return result;
}
```

# Использование зависимых имен

```
template<class T>
struct Array {
    typedef T value_type;
    ...
private:
    size_t size_;
    T * data_;
};

template<class Container>
bool contains(Container const& c,
              typename Container::value_type const& v);

int main()
{
    Array<int> a(10);
    contains(a, 5);
    return 0;
}
```

# Использование функций для вывода параметров

```
template<class First, class Second>
struct Pair {
    Pair(First const& first, Second const& second)
        : first(first), second(second) {}
    First first;
    Second second;
};

template<class First, class Second>
Pair<First, Second> makePair(First const& f, Second const& s) {
    return Pair<First, Second>(f, s);
}

void foo(Pair<int, double> const& p);

void bar() {
    foo(Pair<int, double>(3, 4.5));
    foo(makePair(3, 4.5));
}
```

# Компиляция шаблонов

- Шаблон независимо компилируется для каждого значения шаблонных параметров.
- Компиляция (*инстанцирование*) шаблона происходит в точке первого использования — *точке инстанцирования шаблона*.
- Компиляция шаблонов классов — ленивая, компилируются только те методы, которые используются.
- В точке инстанцирования шаблон должен быть полностью определён.
- Шаблоны следует определять в заголовочных файлах.
- Все шаблонные функции (свободные функции и методы) являются *inline*.
- В разных единицах трансляции инстанцирование происходит независимо.

## Резюме про шаблоны

- Большие шаблонные классы следует разделять на два заголовочных файла: объявление (`array.hpp`) и определение (`array_impl.hpp`).
- Частичная специализация и шаблонные параметры по умолчанию есть только у шаблонов классов.
- Вывод шаблонных параметров есть только у шаблонов функций.
- Предпочтительно использовать перегрузку шаблонных функций вместо их полной специализации.
- Полная специализация функций — это обычные функции.
- Виртуальные методы, конструктор по умолчанию, конструктор копирования, оператор присваивания и деструктор не могут быть шаблонными.
- Используйте `typedef` для длинных шаблонных имён.