# Advanced C++

#3 Функции.
Структуры.
Классы.

# Menu

- Функции
- Структуры
- Классы

# Функции

# Функции

```
int square(int x)
{
    int tmp = x * x;
    return tmp;
}
```

# Конвенции вызова

```
void foo1(int x, int y, int z, int a)
{
}

void foo2(int x, int y, int z, int a)
{
}

void bar1()
{
    foo1(1, 2, 3, 4);
}

void bar2()
{
    foo2(5, 6, 7, 8);
}
```

# Конвенции вызова(fastcall)

```
g++ -c test.cpp -o test.o -O0 -m32

objdump -d test.o
```

Через стек 1 и 2 аргумент, остальные -- через регистр

```
000005c8 <_Z4bar1v>:
 5c8:   6a 04                   push   $0x4
 5ca:   6a 03                   push   $0x3
 5cc:   ba 02 00 00 00          mov    $0x2,%edx
 5d1:   b9 01 00 00 00          mov    $0x1,%ecx
 5d6:   e8 b5 ff ff ff          call   590 <_Z4foo1iiii>
 5dd:   c3                      ret
```

# Конвенции вызова (cdecl)

```
g++ -c test.cpp -o test.o -O0 -m32

objdump -d test.o
```

Все 4 аргумента через стек

```
000005eb <_Z4bar2v>:
 5eb:    6a 08                   push   $0x8
 5ed:    6a 07                   push   $0x7
 5ef:    6a 06                   push   $0x6
 5f1:    6a 05                   push   $0x5
 5f3:    e8 b3 ff ff ff          call   5ab <_Z4foo2iiii>
 5fd:    c3                      ret
```

# Inline functions

```
inline void foo()
{
}

// ms vc
__forceinline void foo()
{
}

// gcc
__attribute__((always_inline)) void foo()
{}
```

# Функции

```
type f([type] x, [type] y) {
    return;
}
```

**void* -- указатель на какую-то память, под которой не понятно, что лежит (какой тип данных).**

# Указатель на функции

Объявленная функция хранится в памяти в виде набора инструкций, которые при вызове загружаются в оперативную память.

```
int f(double, char*);
```

Можно взять адрес функции: `type p = &f;`

Но какой тип у переменной `p`?

```
int (*p)(double, char*) = &f;
```

Пример использования -- компаратор для быстрой сортировки.

```
void qsort(int* a, int count, bool(*cmp)(int, int))
```

```
Разыменовывать уже не надо -- в теле используем просто cmp(val1, val2);
```

# Функции с переменным числом аргументов

```
void f(int x, …);
```

Подключаем <cstdarg>

https://en.cppreference.com/w/cpp/header/cstdarg

Аргументы по умолчанию:

```
void f(int x, char y = 'a');
```

Аргументы по умолчанию в конце списка

Это все легаси языка C.

# Перегрузка (overload)

**Терминология: перегрузка не потому, что сложно, а потому, что существуют разные варианты разного (overload). Несколько функций с одинаковым именем, но разной сигнатурой.**

```
void f(int x);
Int f(char x, int y);
double f(double z);

f('a'); ?

Overloading resolution rules стандарта.
```

# Перегрузка (overload)

```
void print(bool x)
{
    std::cout << (x ? "true" : "false") << std::endl;
}

void print(const std::string& x)
{
    std::cout << "string" << std::endl;
}

print("hello!"); // 2 const char* приводится к bool
```

# Overloading resolution rules

1. Вызов в точности той сигнатуры для того типа, который мы передали
2. Далее идет casting к тем типам, которые имеются
3. Сначала integer promotions (расширение целочисленных типов, т.е. Между int-ами разного размера)
4. Если не получается, то пытается к любому типу
5. Поиск преобразований среди кастомных (пользовательских) типов

```
f(3.14); // double, CE
```

(Т.к. Литералы с дробной точкой считаются по умолчанию double)

```
f(3.14f); // float, точное соответствие
```

# Overloading resolution rules

```
void f(int x, int y);
void f(int x, ...);
```

**Более общий принцип:**

**“Если есть специализированная версия -- выберем именно ее.”**

# Подробнее про операторы new и delete

```
int *p =  new int(...);
```

**Если с помощью `delete p;` удалить не выделенную память -- это формально UB, на деле -- крах программы.**

**Как и не удаление выделенной памяти приводит к memory leak.**

**В случае с массивами:**

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

delete[] p; // correct
delete  p; // UB
```

# Подробнее про операторы new и delete

```
int *p =  new int[100];
int *pp =  new int[100];

delete[] p, pp; // incorrect, pp не удалится
```

**С помощью `delete и new` мы получаем возможность аллоцировать данные из кучи и, соответственно, передавать их в функции и иметь возможность их поменять в процессе работы с ними из функции.**

# Function call operator

**Любая функция, на самом деле, представляет собой использование оператора (), который вызывает функцию и принимает список comma-separated аргументов (*argument-expression-list*), которые впоследствии передаются вызываемой функции.**

**Выдержка из стандарта:**

*argument-expression-list* может быть пустым. До C++ 17 порядок вычисления выражения функции и выражений аргументов не определен и может возникать в любом порядке. В C++ 17 и более поздних версиях выражение функции вычисляется перед любыми выражениями аргументов или аргументами по умолчанию. Выражения аргументов вычисляются в неопределенной последовательности.

# Ссылки

```
int a = 1;
int b = 2;
int* ptr = nullptr;
ptr = &a;
ptr = &b;
int& ref; // Ошибка
int& ref = a; // ref ссылается на a
ref = 5; // Теперь a == 5
ref = b; // ref не стала указывать на b,
         // мы просто скопировали значение из b в a
ref = 7; // a == 7, b == 2
```

```
int a = 2;
int* ptr = nullptr;
int*& ptrRef = ptr; // ptrRef ссылается на ptr
  ptrRef = &a; // теперь ptr хранит адрес a
```

# Ссылки

**scanf(“__”, &x);**

**Функция swap(a,b);**

```
void swap(int x, int y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

# Ссылки

**scanf(“__”, &x);**

**Функция swap(a,b);**

**В Си:**

```
void swap(int* x, int* y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}
```

# Ссылки

**scanf("__", &x);**

**Функция swap(a,b);**

**В Си:**

```
void swap(int& x, int& y) {
    int tmp = x;
    x = y;
    y = tmp;
}
```

- Копирования не происходит, при попытке изменения объекта будет ошибка
- Большие объекты выгодней передавать по ссылке, маленькие - наоборот
- Следует использовать, если надо изменить объект внутри функции

**В Java и Python такой семантики нет. В C++, присвоение -- создание нового инстанса.**

# Ссылки

```
scanf("__", &x);
```

**Константные ссылки:**

```
void foo(const int& x)
{
    x = 3; // ошибка компиляции
}

void bar(const BigObject& o) { ... }
```

Пару слов про константные указатели:

```
const int* p = pp;
```

# Ссылки

Не путать с указателем, который является константой!

```
int* const p = pp // нельзя менять указатель

const int * const p = pp // нельзя менять ни указатель, ни то,
// на что он указывает
```

# Лямбда-функции

```
auto lessThen3 = [](int x) { return x < 3; };

if (lessThen3(x)) { ... }
```

**Синтаксис**

```
[список_захвата](список_параметров) { тело_функции }

[список_захвата](список_параметров) ->
тип_возвращаемого_значения { тело_функции }
```

# Лямбда-функции

```
int x = 5;

int y = 7;

auto foo = [x, &y]() { y = 2 * x };

foo();
```

# Лямбда-функции

```
[] // без захвата переменных из внешней области видимости
[=] // все переменные захватываются по значению
[&] // все переменные захватываются по ссылке
[x, y] // захват x и y по значению
[&x, &y] // захват x и y по ссылке
[in, &out] // захват in по значению, а out — по ссылке
[=, &out1, &out2] // захват всех переменных по значению,
// кроме out1 и out2, которые захватываются по ссылке
[&, x, &y] // захват всех переменных по ссылке, кроме x,
 // которая захватывается по значению
```

# Лямбда-функции

```
int x = 3;
auto foo = [x]() mutable
{
    x += 3; // CE
    ...
}
```

# Лямбда-функции

Можно также реализовать с помощью перегрузки оператора () в классах (рассмотрим позже).

# Структуры и классы

# Структуры

```
struct User
{
    std::string name;
    std::string email;
};

const User user =
    { "Bob", "bob@mail.ru" };

std::cout << user.name;
```

# Структуры

Нужен поиск среди имен пользователей. Что эффективнее?

```
User users[N];
```

**или**

```
struct Users
{
    std::string name[N];
    std::string email[N];
};
```

# Модификаторы доступа

```
struct A
{
public:
    int x; // Доступно всем
protected:
    int y; // Наследникам и объектам структуры (класса)
private:
    int z; // Только объектам структуры (класса)
};

A a;
a.x = 1; // ок
a.y = 1; // ошибка
a.z = 1; // ошибка
```

# Модификаторы доступа

**В C++ struct от class отличаются только модификатором доступа по умолчанию. По умолчанию содержимое struct доступно извне (public), а содержимое class - нет (private).**

```
class A
{
    int x; // private
};

struct B
{
    int x; // public
}
```

# Методы

```
struct User
{
    void serialize(Stream& out)
    {
        out.write(name);
        out.write(email);
    }

private:
    std::string name;
    std::string email;
};
```