Национальный исследовательский университет
«Высшая Школа Экономики»
Факультет компьютерных наук

Лекция 4

Лекторы: Роман Халкечев, Лунев Кирилл

Цели лекции и семинара

- Познакомиться с функциями в C++, перегрузкой функций
- Научиться работать с указателями и ссылками
- Изучить способы создания псевдонимов для типов

План лекции

- Функции: определение, объявление, сигнатура
- Использование заголовочных файлов
- Области видимости
- Указатели
- Ссылки
- Перегрузка функций
- Псевдонимы типов

Функции

Функция - блок кода, выполняющий определенные операции

```
int sum(int num1, int num2) {
    return num1 + num2;
}

void sayHello() {
   std::cout << "Hello!" << std::endl;
}
```

Функции полезны для *инкапсуляции* основных операций в едином блоке, который может многократно использоваться.

Вызов функции

Что будет выведено на экран?

```
int result = sum(3, 5);
result = sum(result, sum(result, result));
cout << result << endl;
```

После оператора `return` в памяти создается ячейка с возвращенным значением. Вызывающая функция сама решает как распорядиться этим значением.

Определение функции

Определение функции - код реализации функции.

Определение - это то, что нужно **линковщику**, чтобы связать вызовы функций (ссылки на функции) с самой сущностью функции.

```
int sum(int num1, int num2) {
    return num1 + num2;
}
```

Каждая функция должна быть определена ровно один раз.

Рекурсия - вызов функции из неё же самой

```
int s(int lhs, int rhs) {
    if (!lhs) {
        return rhs;
    } else if (lhs < 0) {
        return s(lhs + 1, rhs - 1);
    } else {
        return s(lhs - 1, rhs + 1);
    }
}
```

Значение аргумента по умолчанию

```
void DrawSquare(double size = 5., string color="red") {
    ...
}

DrawSquare(10., "blue"); // OK

DrawSquare(15.); // OK

DrawSquare("yellow"); // Not OK
```

Объявление функции

Прототип функции (объявление функции) - называется объявление, не содержащее тела функции, но указывающее:

- имя функции,
- арность,
- типы аргументов
- возвращаемый тип данных.

Прототип - необходим компилятору, для проверки параметров при вызове.

```
int sum(int num1, int num2);
```

Прототип описывает **как** работать с функцией.

Вопрос

Сколько различных прототипов представлено?

```
int sum(int num1, int num2);
int mySum(int num1, int num2);
int sum(int number1, int number2);
double sum(double num1, double num2);
int sum(const int num1, const int num2);
int sum(int, int);
```

Пример

```
void f();

int main() {
   f();
}

void f() {
  cout << "Hello!" << endl;
}
```

Заголовочные файлы

Разработка ПО обычно идет по следующему сценарию:

- определения функций располагаются в **файле исходного кода** (.cpp)
- объявления функций выносятся в **заголовочный файл** (.h)
- исходные коды компилируются
- библиотека - набор откомпилированных файлов реализации
- другие разработчики получают библиотеку и заголовочный файл, конечные пользователи - библиотеку и исполняемый файл

Заголовочные файлы

Преимущества такого подхода:

- Уменьшение времени компиляции
- Код становится более организованным
- Разделение *интерфейса* от *реализации*

Пример

```
//main.cpp
#include "mymath.h"
int main() {
    int x = 10;
    int xSquared = square(x);
    int xCubed = cube(x);
}
```

```
g++ mymath.cpp -c
g++ main.cpp mymath.o
```

```
// mymath.h
double square(double x);
double cube(double x);
```

```
//mymath.cpp
double square(double x) {
    return x * x;
}
double cube(double x) {
    return x * x * x;
}
```

Области видимости (локальные переменные)

- Область видимости переменной определяет участок программы, где переменная известна и доступна.
- Локальная переменная видна внутри блока, где она объявлена
- Глобальные переменные известны на протяжении всей работы программы и доступны из всех функций

Области видимости (пример)

```
int x = 3;
int func(int y) {
     int x = 5;
     for (int i = 0; i < 10; ++i) {
       y += i;
     }
     {int y = 5;}
     if (y < 10)
       int a = 5;
     else
       int a = 100;
     cout << x << y << a << i << ::x << endl; // what's wrong?
}
```

Ключевое слово static

Ключевое слово статик может использоваться:

- для глобальных переменных и функций - это делает их видимыми только внутри текущего файла
- для локальных переменных - переменная остается в памяти до конца программы

Другие применения слова static будут разобраны на следующих лекциях

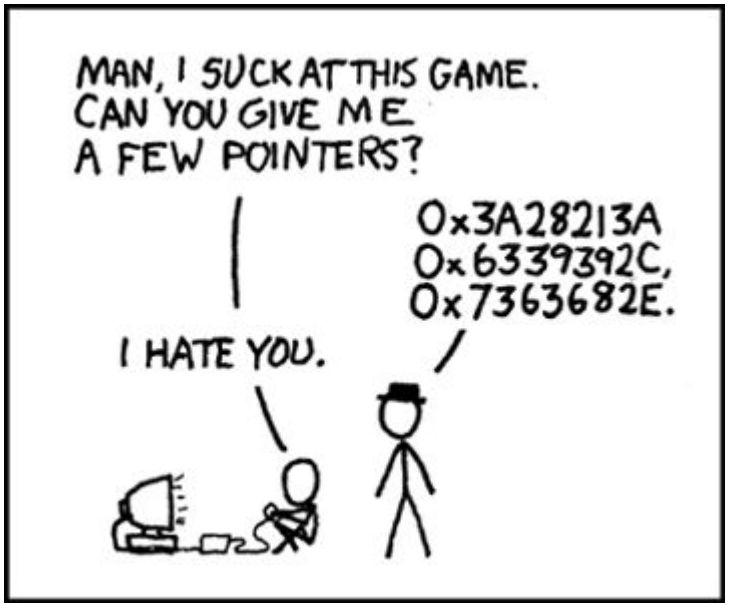
Что будет выведено в терминал?

```
void printLine() {
    static size_t counter = 0;
    cout << "Line number " << ++counter << endl;
}

int main() {
    for (size_t i = 0; i < 10; ++i) {
        printLine();
    }
}
```

Указатели

Переменная, которая хранит адрес другой ячейки памяти называется **указателем**.

Указатель жестко связан с **типом данных**.



Указатель = ссылка на ячейку + способ интерпретировать ячейку.

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

Указатели

```
int x[10];
int* z = x + 3;
++z;

int* a, b; // b - типа int


int* n = 0;
```

Разыменование указателя

Чтобы получить значение, на которое указывает указатель, нужно воспользоваться оператором разыменовывания

```
int a = 10;
int* pa = &a;
int b = *pa;
```

Передача аргументов функции по значению

```
void foo(int x) {
    x++;
}

int main() {
    int y = 3;
    foo(y);
    cout << y << endl; // y == 3
}
```

Изменение аргумента внутри функции

```
void bar(int* x) {
    (*x)++;
}

int main() {
    int y = 3;
    bar(&y);
    cout << *y << endl; // y == 4
}
```

Ссылочная переменная (псевдоним переменной)

```
int x = 10;
int& r = x; //reference
```

```
int* p = &x; //pointer
```

Ссылочные переменные

Отличия от указателей:

- указатель может быть нулевым
- ссылка всегда связана с одним и тем же объектом

Когда это нужно:

- Короткий псевдоним: заменить часто используемую, длинную переменную на короткий псевдоним.
- Использование в функциях: передача по ссылке

Ссылочные переменные

```
void baz(int& x) {
   x++;
}

int main() {
    int y = 3;
    baz(y);
    cout << y << endl; // y == 4
}
```

Пример: функция swap

```
void swap(int& lhs, int& rhs) {
    int temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

Задача

Напишите функцию swap на указателях:

```
void swap(int* lhs, int* rhs) {
    ...
}
```

Проблема ссылок

По вызову неизвестно изменится ли параметр или нет. Поэтому часто для неочевидных случаев так писать запрещают.

Вместо этого используют указатели.

Модификатор const для указателей

```
int x[10];
```

1. `const int * p1 = x;` - указатель на интовую константу
2. `int const * p2 = x;` - указатель на константный инт
3. `int * const p3 = x;` - константный указатель на инт
4. `const int * const p4 = x;` - константный указатель на константный инт

Так можно:

```
p2++;
(*p3)++;
```

Так нельзя:

```
(*p2)++;
p3++;
```

Модификатор const для ссылок

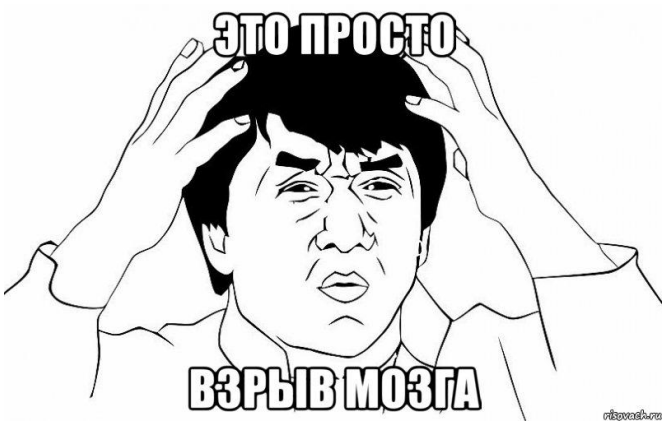
```
int a = 5;


const int & p1 = a;
int const & p2 = a;
int & const p3 = a; // NOT OK


const int& p4 = 5;
int& p5 = 5; //NOT OK
```

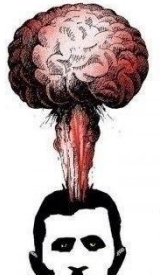
Передача аргумента

Если аргумент является сложным типом и не изменяется внутри функции, нужно передавать его следующим образом:

```
void f(const string& str) {
    ...
}
void g(const int& number) {
    ...
} // но так лучше не делать
```

...


ЭТО ПРОСТО
ВЗРЫВ МОЗГА



Перегрузка функций

Сигнатура функции - типы и порядок аргументов функции.

Тип возвращаемого значения не учитывается.

Перегрузка функции - возможность определить функцию с тем же именем, но с другой сигнатурой.

```
int sum(int num1, int num2);//sum с сигнатурой (int, int)

double sum(double num1, double num2);//sum с сигнатурой
                                     (double, double)
```

Перегрузка функций

```
void print(double length, double width);
void print(int year, int month);
void print(const std::string& str);

print(3.0, 5.0);
print("abcd");

print(3, 5.0); // ?
```

Перегрузка функций

```
void f(int x, int y = 3) {
    cout << "f x y" << endl;
}

void f(int x) {
    cout << "f x" << endl;
}
f(3); // Error
```

Перегрузка функций

Примеры использования:

- перегрузка функции чтения/записи

```
void load(const std::string& filename);
void load(std::ifstream& stream);
```

- перегрузка операторов

```
ostream& operator<<(ostream& stream, const std::vector<int>&
vector);
vector<int> operator+(const vector<int>& lhs,
                      const vector<int>& rhs);
```

Псевдонимы типов

Директива `typedef` позволяет задать синоним для встроенного либо пользовательского типа данных

```
typedef double Points;
typedef std::vector<std::string> StringVector;
typedef std::pair<std::string, double>> WordCount;
typedef std::vector<WordCount> WordCountVector;

WordCountVector CalculateWordCountVector(const StringVector&
texts) { … }
```

Псевдонимы типов

Использование псевдонимов типов:

- Укорачивает запись сложных типов
- Позволяет дать сложному типу осмысленное имя

Псевдонимы типов

Другой способ (c++11) дать типу псевдоним:

```
using StringVector = std::vector<std::string>;
```

Этот способ не хуже `typedef`.

А на самом деле даже лучше.

Чем лучше - расскажу на следующей лекции.

Домашнее задание

- Прочесть главы 7-8 в книге Стивена Прата «Язык программирования C++»
- Дорешать задачи с семинара

Конец

Вопросы?