Основы C++

Пример “Hello World” на языке C++

```
//Подключение модулей/библиотек (директивы препроцессора)
#include<iostream> //Вывод в поток
#include<string> //Строковый тип данных string
//Функция с аргументом по умолчанию
void printStr(std::string str = ""){
  //Цикл проходит по каждому символу строки
  for(int i = 0; i < str.size(); ++i){
    std::cout  << str[i]; //Вывод i-го символа строки в поток
  }
}
//Функция main - точка входа в программу
int main(){
  std::string str = "Hello World"; //Инициализации переменной типа string
  printStr(str); //Вызов функции
  return 0; //Возвращаемое значение функции
}
```

Типы данных

- Типы данных C: **char**, **int**, **float**, **void**
- **double** - 64-битный тип данных для вещественных чисел
- **bool** - логический тип данных. Принимает значение **true** или **false**

- Модификатор **long** можно применить к **int** и **double**
- Модификатор **unsigned/signed** можно применить к **int** и **char**
- Модификатор **const** делает переменную неизменяемой

- *Размер типа данных может меняться в зависимости от ОС*
- Типы фиксированного размера (fixed width): **int8_t**, **int16_t**, **uint64_t...**

Инициализация переменных в C++

```cpp
//Инициализация переменных
int var1 = 333;
int var2 = 1.11;

int var3; //Создание неинициализированной переменной
var3 = 222; //Присваивание значения

//Инициализация в функциональном виде
int var4(333);
int var5(1.11);

//Инициализация через фигурные скобки с проверкой значений
int var6{333};
int var7{1.11}; //Ошибка компиляции double → int
```

Функции - сигнатура

Сигнатура записи функций:

<возвращаемый тип> <имя функции>([набор аргументов]){ ... }

Пример сигнатуры функции:

```
double myFunction(char symbol, int value = 10)
```

double - возвращаемый тип

myFunction - имя функции

Аргументы: char symbol - символьный тип, имя symbol; int value - целочисленный тип, имя value. Имеет значение по умолчанию 10.

Аргументы по умолчанию идут в конце.

Функции - тело

В теле функции записываются инструкции для выполнения. Инструкции отделяются символом ;

Для возврата значения из функции используется оператор **return**

```
double sum(double x){
  double y;
  std::cin >> y;
  return x + y;
}
```

Функция main

Функция main является точкой входа в программу - функция, которая будет вызвана при запуске программы.

Сигнатуры функции:

- int main() { ... }
- int main(int argc, char* argv[]) { ... }

int argc - количество аргументов запуска программы. 0-й аргумент - строка с командой запуска.

char* argv[] - аргументы запуска

Раздельная компиляция (1)

Для читаемости и модульности код разделяется на файлы с объявлением (.h) формата и с определением (*.cpp).

```
//sum.h
#ifndef SUM_H
#define SUM_H

double sum(double x, double y);

#endif
```

```
//sum.cpp
#include "sum.h"
#include<iostream>

double sum(double x, double y){
  double z;
  std::cin >> z;
  return x + y + z;
}
```

Раздельная компиляция (2)

Для сборки проекта используется **g++:**

1) **g++ main.cpp sum.cpp** - соберет проект, программа будет названа по умолчанию “a”

2) **g++ -c main.cpp sum.cpp** - флаг **-c** сообщает, что надо создать только объектные файлы (*.o)

g++ main.o sum.o -o sum_prog - собирает объектные файлы в исполняемый, флаг **-o** задает имя исполняемому файлу.

Флаг **-g** сохраняет информацию для отладки

```
#include "sum.h"
#include <iostream>

int main(){
  double x,y;
  std::cin >> x >> y;
  std::cout << sum(x,y);
  return 0;
}
```

Циклы

В C++ три вида циклов:

for - цикл выполняется заданное количество раз

while - цикл с предусловием. Выполняется по условие истинно. Условие проверяется сначала.

do while - цикл с постусловием. То же самое, что и цикл **while**, но условие проверяется в конце

```
for(int i = 0; i < 10; ++i){
  //int i = 0 - инициализация цикла
  //i < 10 - выполнение пока true
  //++i - изменения после каждой итерации
}
```

```
int i = 0;
while(i < 10){
  //i < 10 - цикл выполняется пока true
  ++i;
}
```

```
int i = 0;
do{
  ++i;
  //Сначала выполняется тело
  //Потом проверка условия
}while(i < 10);
```

Условный оператор

Условный оператор if проверяет условие на истинность. Если условие истинно, то выполняет последующий код.

Может дополняться не обязательным блоком else, код которого будет выполнен, если условие ложное.

```
int value;
if(value % 2 == 0){
  std::cout << "Even number\n";
}
else{
  std::cout << "Odd number\n";
}
```

Чтение и запись в поток

- Инструменты ввода и вывода находятся в модуле **iostream**
- **std::cin** - объект обеспечивающий доступ к потоку чтения
- **std::cout** - объект обеспечивающий доступ к потоку записи
- Для чтения и записи используются операторы >> и << соответственно
- В отличие от **scanf** и **printf**, **cin** и **cout** автоматически определяют тип
- В модуле **iomanip** находятся манипулятора потока

```
bool a;
std::cin >> a;
std::cout << std::boolalpha << std::setw(6) << std::right;
std::cout << a << '\n';
```

Указатели

- Переменная, хранящая адрес некоторой ячейки памяти размером под указанный тип данных
- Нулевому указателю не соответствует никакая ячейка памяти
- Для работы с указателем используются операторы:
  - **&** - взятие адреса
  - ***** - получение значения по адресу (разыменование)

```
int value = 333;
int* pointer = 0; //нулевой указатель на int
pointer = &value; //записан адрес переменной value
*pointer = 42; //разыменование по адресу и запись значения 42
std::cout << value; //теперь в value записано 42
```

Нулевой указатель

- В C++ для нулевого указателя используется отдельный тип данных **nullptr_t**
- Использование данного типа данных позволяет избежать ошибок

```
void func(const int& arg){}

int main(){
  int* ptr_1 = 0;
  int* ptr_2 = NULL;
  int* ptr_3 = nullptr;

  func(NULL); //Предупреждение
  func(nullptr); //Ошибка
}
```

Работа с динамической памятью

- Работа с памятью с использованием инструментов языка C:
  - Функции **calloc** и **malloc** - выделение памяти
  - Функция **free** - очистка памяти
- Работа с памятью с использованием инструментов языка C++:
  - Оператор **new** - выделение памяти
  - Оператор **delete** - очистка памяти
- **new** не только выделяет память, а также проводит инициализацию

Выделение и очистка памяти

При выделении памяти можно сразу задать начальное значение

```
int * value = new int{333}; //Выделение и инициализации память
std::cout << "Address " << value << " Value " << *value << '\n';
delete value; //Очищение памяти

delete value; //Двойное очищение может вызвать SegFault

int * ptr = nullptr;
delete ptr; //Можно очищать нулевой указатель
```

Выделение и очистка памяти под массив

- При выделении памяти в **[ ]** указывается количество ячеек для выделения
- Оператор **new** автоматически определит кол-во необходимой памяти исходя из указанного типа данных
- При удалении необходимо указывать **[ ]**, иначе будет удален только первый элемент

```
int main(){
  int n;
  std::cin >> n;
  int* arr = new int[n];
  for(int i = 0; i < n; ++i){
    std::cout << arr[i] << ' ';
  }
  std::cout << '\n';
  delete [] arr;
}
```

Ссылки в C++

- Исправляют некоторые недостатки указателей
- По факту являются оберткой над указателем
- Уменьшают количество операторов разыменования и взятия адреса

- Для создания переменной ссылки, после типа данных необходимо добавить символ **&**

```
void swap(int& a, int& b){
  int temp = a;
  a = b;
  b = temp;
}
int main(){
  int k = 10;
  int m = 20;
  swap(k,m);
  std::cout << k << ' ' << m << '\n';
  //Output: 20 10
  return 0;
}
```

Различия ссылок и указателей (1)

- Ссылка не может быть не инициализированной

```
int* pointer;   //OK
int& reference; //Ошибка
```

- У ссылки нет нулевого значения

```
int* pointer = 0;   //OK
int& reference = 0;  //Ошибка
```

- Нельзя создать массивы ссылок

```
int* pointer[10];    //OK
int& reference[10];  //Ошибка
```

Различия ссылок и указателей (2)

- Ссылку нельзя переинициализировать - т.е. сменить адрес

```
int a = 10;
int b = 10;
int* pointer = &a; //указывает на a
pointer = &b; //теперь указывает на b
int& reference = a; //является ссылкой на a
reference = b; //переменной a присваивается значения b
```

- Нельзя получить адрес ссылки или ссылку на ссылку

```
int value = 10;
int& reference = value; //ссылка на value
int* p_ref = &reference; //указывает на value
int& reference2 = reference; //ссылка на value
int&& r_ref = reference; //ошибка
```

Константные ссылки

- Ссылка сама по себе является неизменяемой

```
int a = 10;
//Эквивалентные записи
int& const c_ref1 = a;
int const& c_ref2 = a;
c_ref2 = -10; //Ошибка, нельзя менять константу
```

- Позволяет избежать копирования переменных при передаче в функцию, и запрещает изменение аргументов внутри функции

```
Point2D midSegmentPoint(const Segment& segment)
```

Исключения

- Оператор **throw** позволяет выбросить объект ошибки
- В блок **try** помещается код, который может выбросить ошибку
- Блоки catch (минимум 1) ловят определенные ошибки
- Если ошибка не была поймана то программа завершается

```
double myDiv(int x, int y){
  if(y == 0)
    throw std::invalid_argument("Division by zero");
  return x/y;
}
int main(){
  try{
    myDiv(10,0);
  }
  catch(std::runtime_error& e){
    std::cout << "Something wrong\n";
  }
  catch(std::invalid_argument& e){
    std::cout << e.what() << '\n';
  }
  catch( ... ){ //поймать любое исключение
    std::cout << "Unexpected error\n";
  }
}
```