

# Язык программирования C++

## Указатели, ссылки, массивы.

Преподаватели:

Пысин Максим Дмитриевич, ассистент кафедры ИКТ

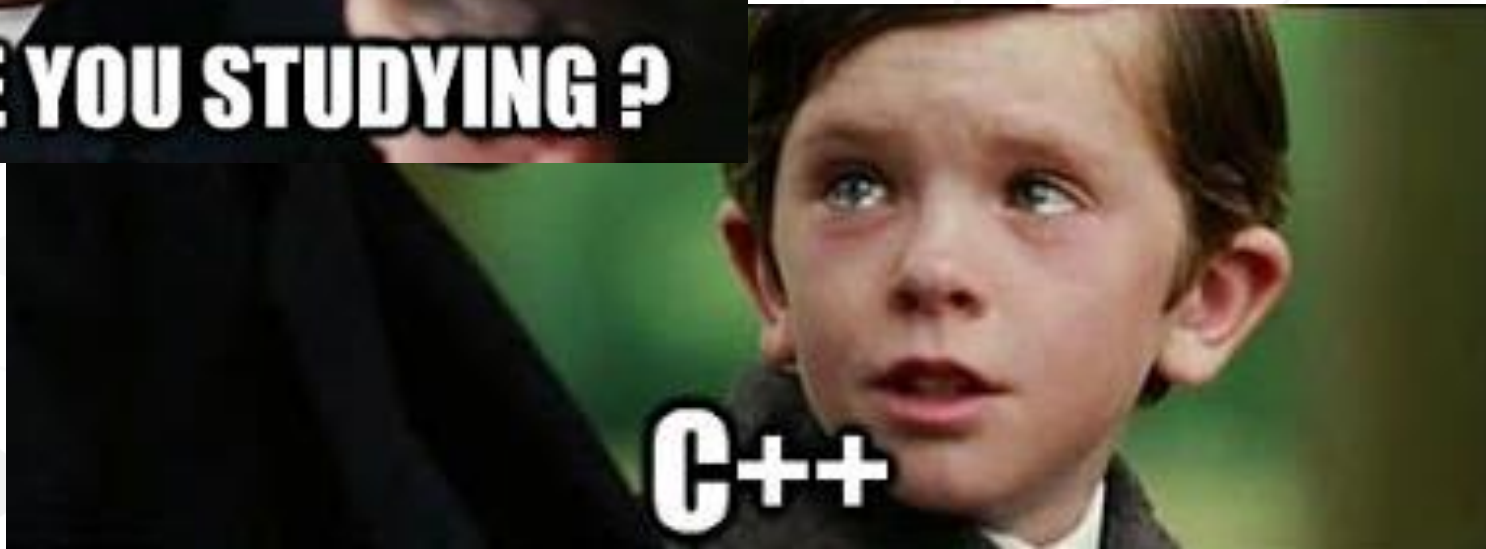
Краснов Дмитрий Олегович, аспирант кафедры ИКТ

Лобанов Алексей Владимирович, аспирант кафедры ИКТ

Крашенинников Роман Сергеевич, аспирант кафедры ИКТ



**WHAT ARE YOU STUDYING ?**



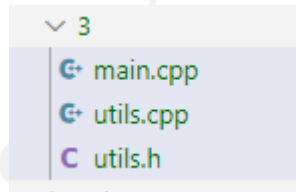
**C++**



# Файловая структура



Файлы:



main.cpp:

```
#include <iostream>
#include "utils.h"
#include <string>

using namespace std;

int main(){
    cout << hellow() << ' ' << world() << endl;
    return 0;
}
```

utils.cpp:

```
#include "utils.h"

std::string hellow(){
    return "hellow";
}

std::string world(){
    return "world";
}
```

utils.h:

```
#ifndef UTILS_H_EXAMPLES
#define UTILS_H_EXAMPLES

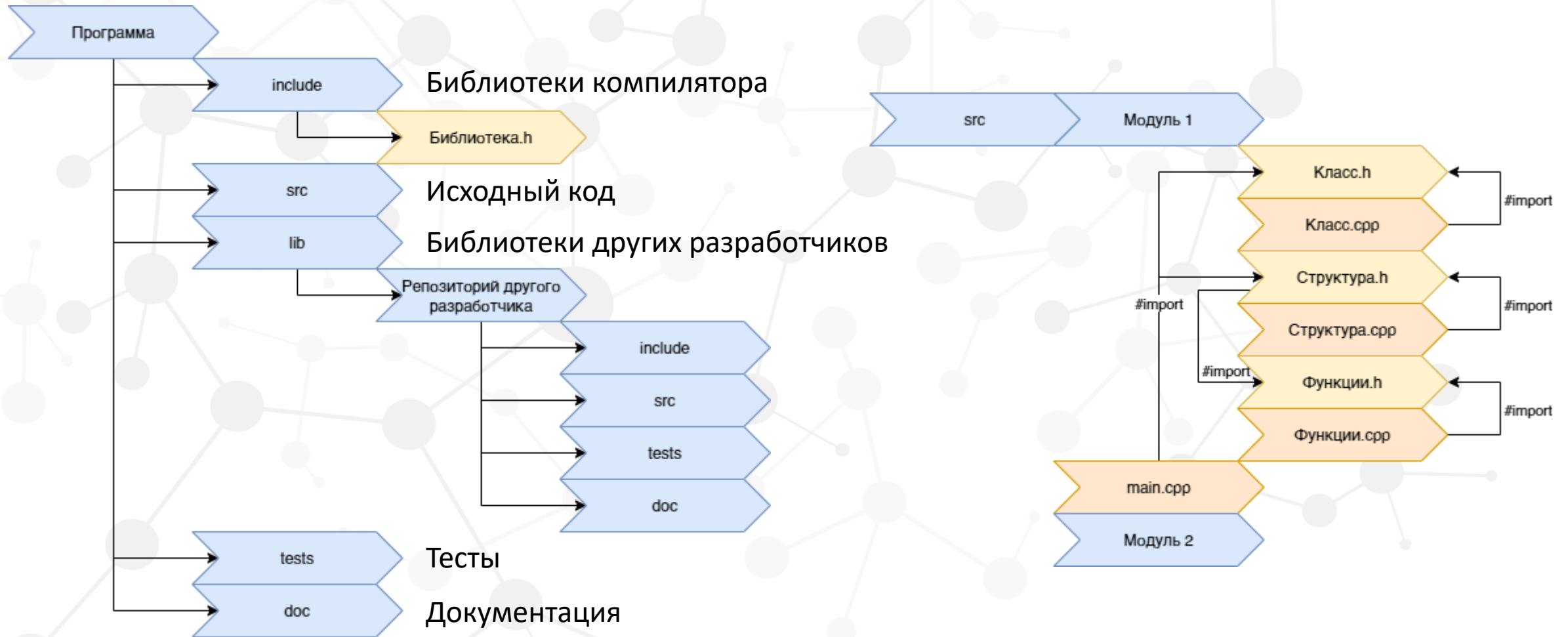
#include <string>

std::string hellow();
std::string world();

#endif
```

Флаги защиты компиляции  
На самом деле условное выражение препроцессора проверяющий был ли выполнен define макроса с именем, и если нет, то делает код, который находится внутри него доступным компилятору.

# Файловая структура



# Статические переменные



Что компилируем?

Куда собираем?

Команда компиляции:

`g++ main.cpp utils.cpp -o main.exe`

Импортированы

Библиотека string

Препроцессор

Компиляция main.cpp

Библиотека iostream

Препроцессор

Компиляция utils.h

Компиляция utils.cpp

Линковщик

Флаги защиты компиляции не позволяют импортировать и компилировать один и тот же код несколько раз



# Область имен(namespace)



main.cpp:

```
#include <iostream>
#include "utils.h"
#include <string>

using namespace std;

int main(){
    cout << hellow() << ' ' << world() << endl;
    return 0;
}
```

Область видимости, это понятие обозначающее некоторый префикс приставляемый к именам переменных, констант, функций и пользовательских типов, отделяемый от них :: а их основной целью служит возможность обеспечить непересекаемость имен.

Пространства имен могут быть вложенные пространства, что отражается на использовании.

Для использования члена пространства имен(в примере функцию) нужно полностью прописать его название с префиксом.

utils.cpp:

```
#include "utils.h"

std::string utils::hellow(){
    return "hellow";
}

std::string utils::world(){
    return "world";
}
```

utils.h:

```
#ifndef UTILS_H_EXAMPLES
#define UTILS_H_EXAMPLES

#include <string>

namespace utils{
    std::string hellow();
    std::string world();
}

#endif
```

# Директива using



Директива `using` позволяет упростить обращения к члену пространства имен, и позволяет опускать указание пространства имен.

Если ее используют с ключевым словом `namespace`, то это означает, что в дальнейшем, все попытки вызвать какую либо функцию, после поиска по файлу и текущему пространству имен, будет производиться поиск в указанном пространстве.

Если ее используют без ключевого слова `namespace`, то тогда после приводят полное название какого либо члена какого либо пространства имен и в дальнейшем его можно будет использовать без префиксов.

utils.h:

```
#ifndef UTILS_H_EXAMPLES
#define UTILS_H_EXAMPLES

#include <string>

namespace super{
    namespace utils{
        std::string hellow();
        std::string world();
    }
}

#endif
```

main.cpp:

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

using namespace std;
using namespace super;
using utils::world;

int main(){
    cout << utils::hellow() << ' ' << world() << endl;
    return 0;
}
```

# Нулевой указатель



Указатель может указывать в пустоту. Фактически пустотой является любой адрес по которому указатель не может обратиться (первый пример, указатель ни на что не указывает, и при обращении по нему выдает ошибку доступа к памяти)

Но как выявить пустой указатель?

К примеру нам нужно в какой либо функции выяснить нужно ли генерировать число, или оно уже есть?

Что является пустотой для указателя?

Указатель на пустоты, это указатель который указывает на 0, синонимами которого является `NULL` или `nullptr`.

`NULL` является наследием C, и является макросом, а вот `nullptr` это специальное обозначение именно нулевого указателя в C++.

```
int main(){
    int* p;
    cout << "&" << p << ": " << *p << endl; // Segmentation
    fault
    return 0;
}
```

```
void generateRandInteger(int* pi){
    if(pi == NULL && pi == nullptr && pi == 0)
        pi = new int {rand()};
}

int main(){
    srand(time(NULL));
    int* pi = NULL;
    // cout << *pi << endl; // Segmentation fault
    cout << "&" << pi << endl;
    pi = generateRandInteger();
    cout << "&" << pi << ": " << *pi << endl;
    return 0;
}
```



# Пользовательские типы данных



Пользовательским типом данных называется любой тип который не является заранее установленным в языке.



# Перечисления



Перечисление, это специальная структура языка которая позволяет хранить только заранее предустановленный набор значений.

Сигнатура:

```
enum <название>{  
    // перечисление названий значений через запятую  
};
```

Перечисление самостоятельно присвоит значения каждому названию, однако вы можете сделать это самостоятельно используя строчку вида: <название значения> = <значение>.

Значением в данном случае может являться только число типа int

```
enum Status{  
    SUCCESS,  
    FAIL,  
    WARNING,  
    ERROR,  
    INFO,  
};  
  
int main(){  
    Status current_status(SUCCESS);  
    cout << current_status << endl;           // 0  
    current_status = FAIL;  
    cout << current_status << endl;           // 1  
    Status* pstatus = &current_status;  
    cout << pstatus << " = " << &current_status << endl; // 0x61fe0c = 0x61fe0c  
    *pstatus = INFO;  
    cout << current_status << endl;           // 4  
    Status& rstatus = current_status;  
    current_status = WARNING;  
    cout << rstatus << endl;                 // 2  
    return 0;  
}
```

# Структуры



Структура является объединением нескольких переменных одним общим именем, при этом каждая из переменных объединенных в структуру доступна для записи и чтения.

Переменные перечисленные в структуре называются полями структуры и доступны для обращения только с использованием структуры.

Сигнатура:

```
struct <название>{  
    <тип поля> <имя поля>;  
    // ...  
};
```

Создание переменной с типом структуры:

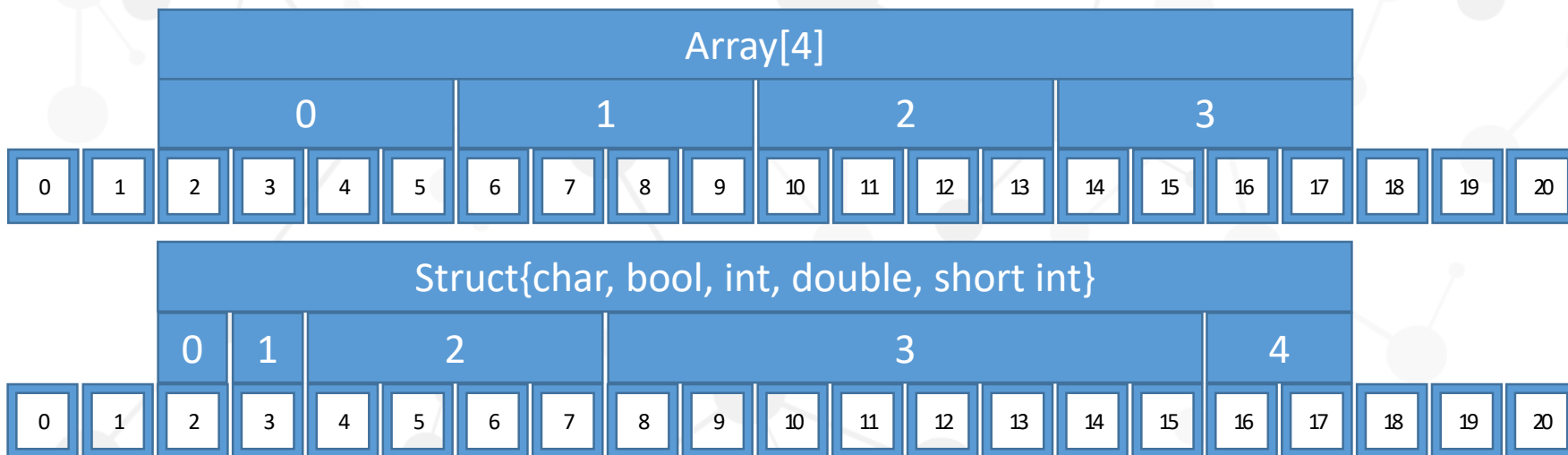
```
<название структуры> <имя переменной>;
```

```
<название структуры> <имя переменной> {<значения полей через запятую>;}
```

Обращение к полям:

```
<переменная структуры>.<наименование поля>;
```

Структура располагается в памяти как один единый блок, переменные располагаются в этом блоке строго друг за другом в порядке объявления структуре, а общий занимаемый структурой объем памяти равен сумме объемов памяти требуемых на хранения каждого из полей структуры по отдельности.



# Пример структуры



```
struct Person{
    string name;
    int day;
    int month;
    int year;
};

int main(){
    Person first;
    first.name = "Михаил";
    first.day = 10;
    first.month = 3;
    first.year = 1998;
    cout << "Имя: " << first.name << "\nДата рождения: " << first.day << '.' << first.month << '.' << first.year
    << endl;
    // Имя: Михаил
    // Дата рождения: 10.3.1998
    Person second {"Дмитрий", 12, 5, 1999};
    cout << "Имя: " << second.name << "\nДата рождения: " << second.day << '.' << second.month << '.' << second.
    year << endl;
    // Имя: Дмитрий
    // Дата рождения: 12.5.1999
    return 0;
}
```

# Пример структуры



```
enum Month{
    JANUARY,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
};

struct Date{
    int day;
    Month month;
    int year;
};

struct Person{
    string name;
    Date birth;
};

int main(){
    Date first_birth {1, JULY, 2000};
    Person first {"Григорий", first_birth};
    cout << &first_birth << endl; // 0x61fe00
    cout << &first.birth << endl; // 0x61fdf0
    cout << "Имя: " << first.name << endl;
    cout << "Дата рождения: " << first.birth.day << '.' << first.birth.month << '.' <<
first.birth.year << endl;
    // Имя: Григорий
    // Дата рождения: 1.6.2000
    Person second {"Алена", {14, DECEMBER, 1993}};
    cout << "Имя: " << second.name << endl;
    cout << "Дата рождения: " << second.birth.day << '.' << second.birth.month << '.'
<< second.birth.year << endl;
    // Имя: Алена
    // Дата рождения: 14.11.1993
    return 0;
}
```

# Функции и структуры



```
void printDate(Date* date){
    cout << date->day << '.' << date->month << '.' << date->year << endl;
}

void printPerson(Person& person){
    cout << "Имя: " << person.name << endl;
    cout << "Дата рождения: " << person.birth.day << '.' << person.birth.month << '.' << person.birth.year << endl;
}

Person* copyPerson(Person* from){
    return new Person {from->name, from->birth};
}

int main(){
    Date first_birth {12, MARCH, 1987};
    Person first {"Николай", first_birth};
    printPerson(first);
    // Имя: Николай
    // Дата рождения: 12.2.1987
    Person* psecond = copyPerson(&first);
    printDate(&psecond->birth); // 12.2.1987
    (*pssecond).name = "Игорь";
    printPerson(*pssecond);
    // Имя: Игорь
    // Дата рождения: 12.2.1987
    return 0;
}
```

При обращении к полям структуры через указатель нужно использовать оператор `->`:  
<указатель на переменную структуры>`->`<наименование поля>;

Этот оператор разыменовывает указатель и обращается к полю, эту операцию можно использовать повторно самостоятельно для обращения через точку:

(`*<указатель на переменную структуры>`).<наименование поля>;



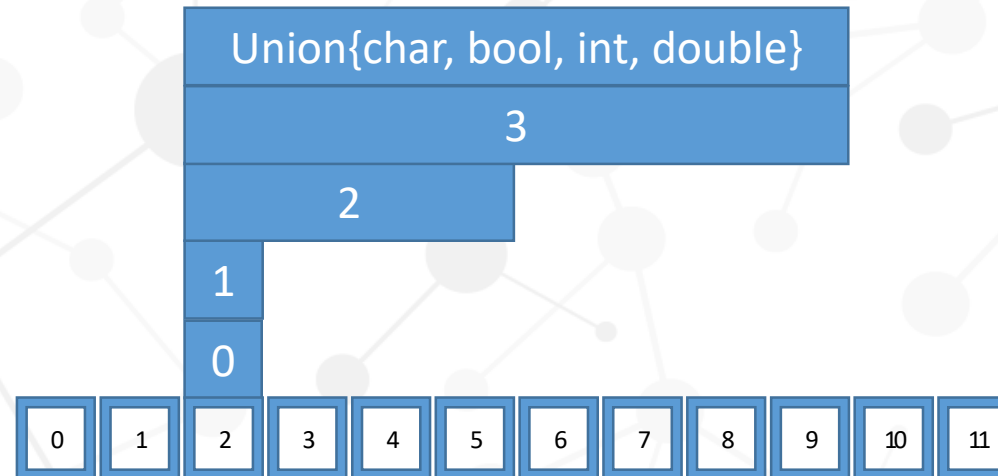
# Объединения



Объединения это специальный пользовательский тип который позволяет хранить данные разного типа по одному и тому же адресу.

Сигнатура:

```
union <название>{  
    <тип> <название поля> = <значение по умолчанию>;  
    <тип> <название поля>  
    // ...  
};
```



- Все объединение занимает памяти как самый большой тип входящий в него
- Обращение к полю объединения происходит так же как к полю структуры.
- На объединения так же как на структуры могут указывать указатели и ссылки.
- Обратиться к значению объединения можно только через поле.

# Пример объединения



```
union Value{
    int i = 0;
    double f;
    char c;
};

void printValue(Value& val){
    cout << val.i << " " << val.f << " " << val.c << " " << endl;
}

int main(){
    Value value;
    cout << sizeof(value) << endl; // 8
    printValue(value); // 0 0 ' '
    value.i = 56;
    printValue(value); // 56 2.76677e-322 '8'
    value.c = 'h';
    printValue(value); // 104 5.13828e-322 'h'
    value.f = 12.434;
    printValue(value); // 104 5.13828e-322 'h'
    cout << value; // Ошибка
    return 0;
}
```

# Псевдонимы



Псевдонимом называется тип который можно использовать для объявления переменных но он не является самостоятельным и за ним скрывается какой то другой тип.

Сигнатура:

**typedef** <для кого является синонимом> <название синонима>;

```
union Value{
    int i = 0;
    double f;
    char c;
};
```

```
struct Date{
    int day;
    int month;
    int year;
};
```

```
typedef int** DoublePointerINT;
typedef unsigned long long int ULLI;
typedef Date* DatePointer;
typedef Value& ReferenceValue;
```

```
int main(){
    DoublePointerINT dpi = new int*[10];
    ULLI big_int = 18'446'744'073'709'551'615LLU;
    DatePointer dp = new Date{1, 1, 2001};
    Value value;
    value.c = 'h';
    ReferenceValue rvalue = value;
    cout << dpi << sizeof(dpi) << endl; // 0xe443808
    cout << big_int << sizeof(big_int) << endl; // 184467440737095516158
    cout << dp->year << endl; // 2001
    cout << rvalue.c << endl; // h
    return 0;
}
```

# Inline Функции



Встраиваемые(они же inline) функции это функция которую компилятор попытается вставить прямо в место ее вызова, таким образом сократив время вызова функции.

Сигнатура:

```
inline <сигнатура функции>;
```

```
string hellow_function() {  
    return "Hellow";  
}
```

```
inline string world_inline_funtion(){  
    return "World";  
}
```

```
int main(){  
    cout << hellow_function() << ' ' << world_inline_funtion() << endl;  
    return 0;  
}
```

Inline функция не дает гарантии, что компилятор в обязательно порядке ее объявление перенес в место ее вызова, но она говорит компилятору, что бы он выполнял ее вызов как можно быстрее любым способом. Функция объявленная как inline навсегда останется inline.

# Аргументы по умолчанию



Параметры передаваемые в функцию могут иметь значение по умолчанию, и не требовать в обязательном порядке своей передачи.

Сигнатура:

```
<тип возвращаемого значения> <название функции>(  
    <параметры не имеющие значения по умолчанию>...,  
    <тип параметра со значением по умолчанию> <название> = <значение>,  
    ...<только параметры имеющие значение по умолчанию>  
);
```

```
void print(string word, string after = " ") {  
    cout << word << after;  
}
```

```
int main(){  
    print("Hellow");  
    print("World", "\n-----\n");  
    // Hellow World  
    // -----  
    return 0;  
}
```

# Неопределенное кол-во аргументов



Помимо аргумента по умолчанию у функции можно указать неопределенное количество аргументов одного типа.

Сигнатура:

```
<тип возвращаемого значения> <название функции>(<тип аргументов> <название первого аргумента>, ...){  
    // тело функции  
}
```

```
double average(double n=0., ...){  
    double *p = &n;  
    double sum = 0, count = 0;  
    while (*p){  
        sum+=(*p);  
        cout << (*p) << " + ";  
        p++;  
        count++;  
    }  
    cout << " = ";  
    return ((sum)?sum/count:0);  
}
```

```
int main(){  
    double a = 10.;  
    double b = 12.;  
    double c = 13.456;  
    int n = 2;  
    int m = 100;  
    cout << average(a, b, c, (double)n, (double)m, 0)  
    << endl;  
    cout << average(a, b, c, 0) << endl;  
    cout << average() << endl;  
    cout << average(n, m, 0) << endl;  
    return 0;  
}
```



# Перегрузка функции



Работа с функциями часто требует использования одной и той же функции для разных типов. Глупо для таких функций иметь разные названия, поэтому в c++ есть механизм перегрузки функции(override) – при котором множество функций имеют одно название но разный список параметров. Тип возвращаемого значения роли не играет.

```
int square(int x){
    return x * x;
}
double square_d(double x){
    return x * x;
}
long square_l(long x){
    return x * x;
}

int main(){
    double dn = 13.456;
    int n = 2;
    long ln = 100;
    cout << "int: " << square(n) << endl;
    cout << "double: " << square_d(dn) << endl;
    cout << "long: " << square_l(ln) << endl;
    return 0;
}
```

```
int: 4
double: 181.064
long: 10000
```

```
void square(int x){
    cout << "int: " << x * x << endl;
}
void square(double x){
    cout << "double: " << x * x << endl;
}
void square(long x){
    cout << "long: " << x * x << endl;
}

int main(){
    double dn = 13.456;
    int n = 2;
    long ln = 100;
    square(n);
    square(dn);
    square(ln);
    return 0;
}
```

# Указатель на функцию



В C++ существуют указатели на функции.

Сигнатура объявления указателя на функцию:

<тип возвращаемого значения> (\*<название указателя>)(<аргументы функции>) = <название функции>;

```
double square(double x) {  
    return x * x;  
}  
void for_each(double* array, const int N, double (*function)(double )){  
    for(int i = 0; i < N; i++){  
        array[i] = function(array[i]);  
    }  
}  
void print(double* array, const int N){  
    for(int i = 0; i < N; i++){  
        cout << array[i] << ' ';  
    }  
    cout << endl;  
}  
int main(){  
    const int N = 10;  
    double array[N] = {0,1,2,3,4,5,6,7,8,9};  
    print(array, N);  
    for_each(array, N, square);  
    print(array, N);  
    return 0;  
}
```

Указатели на функцию можно передавать как аргументы в функцию.

Указатели на функцию могут иметь синонимы.

Могут существовать массивы указателей на функции.

The background is a solid blue color with a complex, abstract network pattern. This pattern consists of numerous light blue circles of varying sizes, which are interconnected by thin, light blue lines. The connections form a dense, web-like structure that fills the entire frame, with some nodes having multiple connections and others being isolated.

**Спасибо за внимание**