

# Достоинства языка C++

- гибкость и компактность языка;
- эффективность, основанную на том, что семантика языка отражает архитектуру компьютера;
- доступность (для любой ОС имеется компилятор с языка в машинно-зависимые коды);
- переносимость. Степень сложности переноса с одной платформы на другую относительно проста по сравнению с другими языками



# Структура программы

```
#include "stdafx.h"
float tax (float) ;
#include <locale>;

int main(int argc, char *argv[])
{ std::setlocale(LC_ALL, "Russian_Russia.1251");
  float cena, tax_amt, total;
  std::cout << "\nСтоимость товара: ";
  std::cin >> cena;
  tax_amt = tax(cena);
  total = cena + tax_amt;
  std::cout << "\nТовар: " << cena<<std::endl;
  std::cout << "Налог: " << tax_amt;
  std::cout << "\nВсего: " << total;
  //_getch();
  return 0;}

float tax (float amount)
{   float rate = 0.065;
  return(amount * rate); }
```

файл для подключения заголовочных файлов, используемых в проекте.

Объявление функций

#include "targetver.h"

#include <iostream>

#include <conio.h>

#include <SDKDDKVer.h>

Описание вспомогательной функции



# Определение времени жизни и области видимости переменных

Время жизни переменной определяется по следующим правилам:

- переменные объявленные на внешнем уровне, всегда имеют глобальное время жизни;
- переменные, объявленные на внутреннем уровне, имеют локальное время жизни. Можно обеспечить глобальное время жизни для переменной внутри блока, задавая ей класс памяти `static` при ее объявлении.

Видимость переменной в программе определяется по правилам:

- переменные, объявленные или определенные на внешнем уровне, видимы от точки объявления или определения до конца исходного файла.;
- переменные, объявленные или определенные на внутреннем уровне, видимы от точки объявления или определения до конца блока;
- переменные из объемлющих блоков, включая переменные, объявленные на внешнем уровне, видимы во внутренних блоках.



# Модификаторы памяти

- **auto** — *автоматическая* переменная. Память выделяется в стеке и при необходимости инициализируется каждый раз при выполнении оператора, содержащего ее определение. Освобождение памяти - при выходе из блока
- **extern** — переменная определяется в другом месте программы.
- **static** — *статическая* переменная. Время жизни — постоянное. Инициализируется один раз при первом выполнении оператора, содержащего определение переменной. В зависимости от расположения оператора описания статические переменные могут быть *глобальными и локальными*.
- **register** — аналогично auto, но память выделяется по возможности в регистрах процессора.



# Пространство имен

Пространство имен namespace - логически объединяет классы с близкой функциональностью предназначено для разрешения конфликтов между именами в разных сборках.

```
namespace Jack  
{ void fetch( ) ;  
  int pal;}
```

```
namespace Jill  
{double fetch ;  
  int pal ;}
```

```
Jack :: pal = 12 ;
```



# Объявления using и директивы using

```
namespace Jill
{ int pal;
  double fetch ;
};
har fetch ;
int main ( )
{
using Jill :: fetch ;
double fetch ;
cin >> fetch ;
cin > > :: fetch ; }
```

```
using namespace System;
std::cout<<"HELLO WORD"<<std::endl
using namespace std;
cout<<"HELLO WORD"<<endl
```

```
Jack :: pal = 3 ;
Jill :: pal = 10 ;
```

```
using Jack :: pal ;
using Jill :: pal ;
pal = 4 ; // Конфликт имен
```



# Константы

Возможно использование трех типов цифровых констант:

- десятичные : 80, 9, 1000000;
- восьмеричные: 060, 07, 02345;
- шестнадцатеричные: 0x80, 0xAFC



# Описание идентификаторов

[<спецификатор класса памяти>] [const]  
<спецификатор типа> <идентификатор>  
[=<начальное значение>];

auto double first=123.34;

const float pi = 3.14;

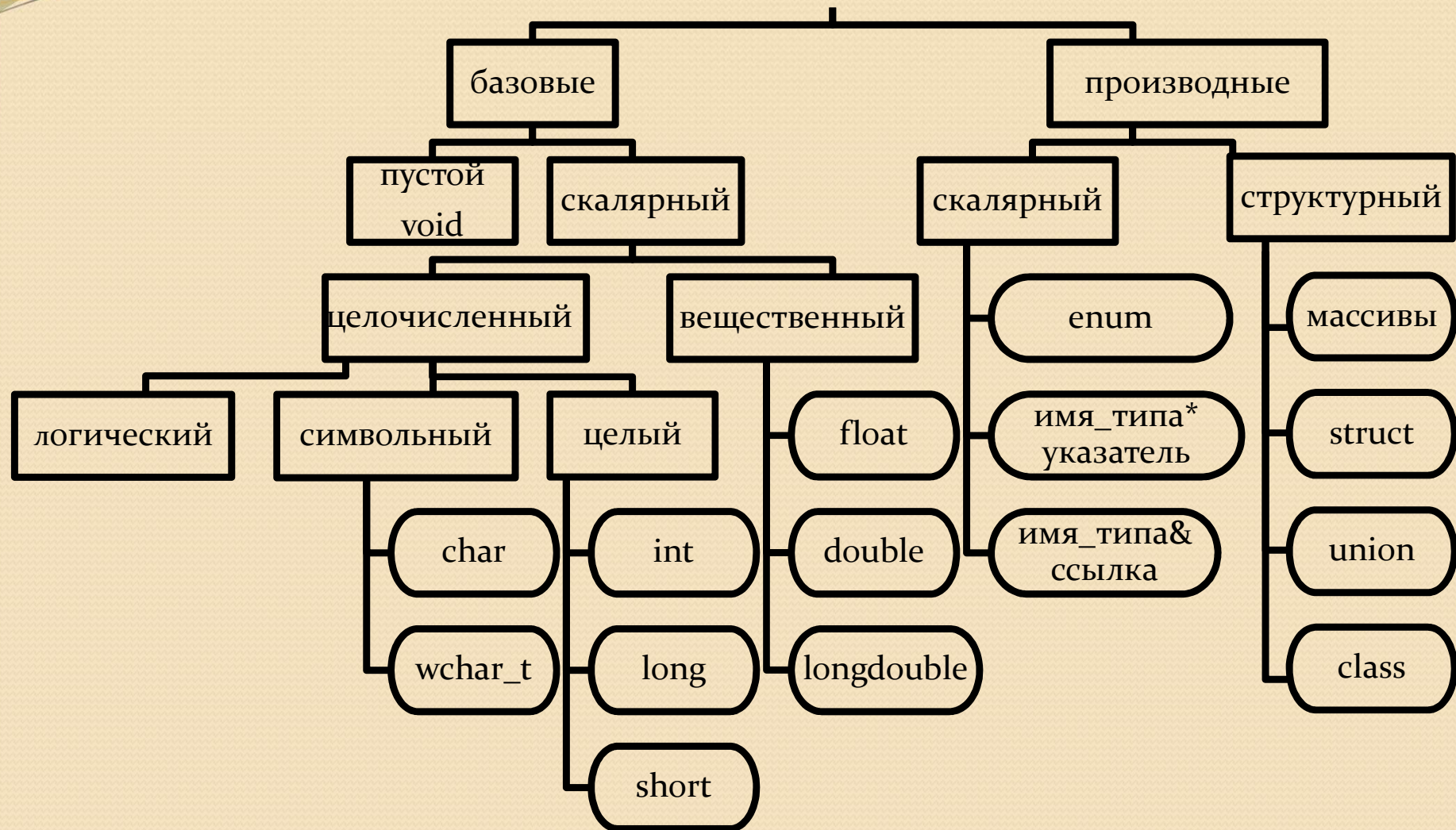
int \*x, z;

float \* near y=NULL;

x = &z; // операция получения адреса



## ТИПЫ данных





## Ссылки

```
int a =3;
```

```
int & another = a;
```

```
another = another + 1; // Переменная a = 4
```

## Перечисление

```
enum Err { ERR_READ, ERR_WRITE, ERR_CONVERT};
```

```
Err error;
```

```
switch (error)
```

```
{
```

```
case ERR_READ: /* операция */ break;
```

```
case ERR_WRITE:      /* операция */ break;
```

```
case ERR_CONVERT:    /* операция */ break;
```

```
}
```



## Массивы

*Объявление:*    `int mass [2];`  
                  `float matr [6] [7];`  
                  `unsigned long Arr3D[4] [2] [8] ;`  
                  `char x[][3] - {{9,8,7},{6,5,4},{3,2,1}};`  
                  `char data = "Это массив";//data[10]="\0"`

*Инициализация:*        `char ArrChar[] = {'W','O','R','L','D'};`  
                              `int Temp[12] = {2, 4, 7, };`

*Обращение:*

через указатель -

`char* pArr = ArrChar;`  
`char Letter = *pArr;`  
`pArr += 3; // указывает на ArrChar[3]`  
`Letter = * (ArrChar + 3) ;`

по индексу -

`Letter = ArrChar[3];`



## В случае многомерны массивов

```
char ArrayOfChar[3][2] = {'W','O','R','L','D','!'};
```

```
char* pArr = (char*)ArrayOfChar;
```

```
pArr += 3; //элемент ArrayOfChar[1] [1]
```

```
char Letter = *pArr;
```

## Передача массива в качестве параметра

```
const int ArSize = 8 ;
```

```
int sum_arr (int arr[ ] , int n ) ;
```

```
int main ( )
```

```
{ int cookies [ArSize ] = { 1 , 2 , 4 , 8 , 16 , 32 , 64 , 128 } ;
```

```
int sum = sum_arr (cookies , ArSize ) ;
```

```
std ::cout << " Всего съеде но печенья : " << sum << " \ n " ;
```

```
return 0;
```

```
}
```

```
int sum_arr ( int arr [ ] , int n )
```

```
{ int total = 0;
```

```
for ( int i = 0; i < n ; i ++ )
```

```
total= total + arr [ i ] ;
```

```
return total ;
```

```
}
```



## Распределение памяти для структуры

```
struct binar{  
    unsigned first :2;  
    unsigned sec :2;  
    unsigned third :3;  
    unsigned forth :4;  
    unsigned fifth :5;  
} int_val;
```

```
int_val.first = 0;  
int_val.sec = 0xf;  
int_val.third = 0;  
int_val.forth = 8;  
int_val.fifth = 0;
```

16																0
0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0

```
union    int_or_long {  
        int i;  
        long l;  
} count;
```



# Операции C++

- **Унитарные**

`c = ++a; // Значение переменных: c = (a+1) ,  
a=(a+1)`

`c = a++; // c=a; a=(a+1)`

- **Бинарные**

`Data+=10; // увеличить на 10 и присвоить`

- **Тернарная :**

*условие ? операнд1 : операнд2;*

Если условие истинно (не равен 0), то вычисляется операнд1 и его значение является результатом операции, иначе вычисляется операнд2

`max = (a > b) ? a : b;`



# Правила преобразования типов

- Операнды `char`, `unsigned char` или `short` преобразуются к `int` по правилам:
  - ❖ `char` расширяется нулем или знаком в зависимости от умолчания для `char`;
  - ❖ `unsigned char` расширяется нулем; `signed char` расширяется знаком;
  - ❖ `short`, `unsigned short` и `enum` при преобразовании не изменяются.
- Если один из операндов имеет тип `long double`, то другой преобразуется к типу `long double`.
- Если один из операндов `unsigned`, другой преобразуется к `unsigned`.
- Тип результата тот же, что и тип участвующих в выражении операндов.
- Операнды преобразуются к тому типу, который имеет наибольший размер.

Возможно и явное преобразование операндов

`int Integer = 54;`

`float Floating = 15.854;`

`Integer = (int) Floating; // явное`

`Integer = Floating; // неявное`



# Функции

Функция — это фрагмент кода, который можно неоднократно вызвать из любого места программы, они позволяют уменьшить избыточность программного кода и повысить его структурированность.

Выделяют описание (объявление и определение) и вызов функции



❖ `void f (int a, int & b)`  
`{a++; b++;}`

❖ `void f (int a, const int & b)`

❖ *Параметры по умолчанию*

`int sum (int a, int b=10)`  
`sum (c,d);` или `sum (x);`

❖ *Переменное число параметров*

```
int sum(int x, ...);  
int main() {  
    std::cout << sum(2, 20, 30) << std::endl;    // 50  
    std::cout << sum(3, 1, 2, 3) << std::endl;    // 6  
    std::cout << sum(4, 1, 2, 3, 4) << std::endl; // 10  
    return 0;  
}  
int sum(int x, ...) {  
    int result = 0;  
    int *p = &x;    // Получаем адрес последнего параметра  
    for(int i=0; i<x; ++i)  
    {  
        ++p;    // Перемещаем указатель на следующий параметр  
        result += *p; // Прибавляем очередное число  
    }  
    return result;  
}
```



## *Суммирование произвольного количества вещественных чисел*

```
double sum(int x, ...);
int main()
{
    std::cout << sum(2, 20.2, 3.6) << std::endl;    // 23.8
    std::cout << sum(3, 1., 2.8, 3.4) << std::endl; // 7.2
    return 0;
}
double sum(int x, ...)
{
    double result = 0.0, *pd = 0;
    int *pi = &x;    // Получаем адрес последнего параметра
    ++pi;           // Перемещаем указатель до приведения типов!!!
    pd = <double *>(pi); // Приведение типов
    for(int i=0; i<x; ++i)
    {
        result += *pd; // Прибавляем очередное число
        ++pd;         // Перемещаем указатель на следующий параметр
    }
    return result;
}
```



# ПЕРЕГРУЗКА ФУНКЦИЙ

```
int sum (int a, int b) {return a+b;}
```

```
// public @sum#qii
```

```
double sum (double a, double b) {return a+b;}
```

```
// public @sum#qdd
```

*zc для char, pi для int\*, pf для float\**

Неоднозначность может появиться при:

- преобразовании типа;
- использовании параметров-ссылок;
- использовании аргументов по умолчанию.



# Неоднозначность перегрузки

- `int sum(int x);`  
`int sum(int x, int y=2);`  
`...std::cout << sum(10, 20) << std::endl; // Нормально`  
`std::cout << sum(10) << std::endl; // Неоднозначность`
- `float sum(float x, float y);`  
`double sum(double x, double y);`  
`std::cout << sum(10.5, 20.4) << std::endl; // Нормально`  
`std::cout << sum(10, 20) << std::endl; // Неоднозначность`
- `void print(char *str);`  
`void print(char str[]);`  
`print("String"); // Неоднозначность`
- `void print(int x);`  
`void print(int &x);`  
`int n = 25;`  
`print(n); // Неоднозначность`



## Правила описания перегруженных функций

- Перегруженные функции должны находиться *в одной области видимости*;
- Перегруженные функции могут иметь *параметры по умолчанию*, при этом значения одного и того же параметра в разных функциях должны совпадать. В различных вариантах перегруженных функций может быть различное количество параметров по умолчанию;
- Функции не могут быть перегружены, если описание их параметров отличается только *модификатором const или использованием ссылки* (например, `int` и `const int` или `int` и `int&`).



# Шаблоны функций

```
template <параметры> заголовок  
{ /* тело функции */ }
```

```
template <class T>  
void printarray (T *array, const int count)  
{for (int i=0; i<count; i++)  
    cout<<array[i]<<" ";  
    cout<<endl;  
}  
int main()  
{const int account=5, bcount=7;  
int a[account]={1,2,3,4,5};  
float b[bcount]={1.1,1.2,1.3,1.4,1.5,1.6};  
printarray (a, account);  
printarray (b, bcount);  
return 0 ;  
}
```



- `template <class T, class U>`  
`T FuncName (U);`
- `template <class T, class T>`  
`T FuncName(T,T);`
- `//Объявление внешней шаблонной функции`  
`template <class T> extern`  
`T* Swap(T* t, int ind1, int ind2);`  
`//Объявление статической шаблонной функции`  
`template <dass T> static`  
`T* Swap(T* t, int ind1, int ind2);`  
`//Объявление встроенной шаблонной функции`  
`template <dass T> inline`  
`T* Swap(T* t, int incU, int ind2);`



# Явная спецификация типов

```
template <class T>
void FuncName(T) { ... } ;
void AnotherFunc(char ch)
{
//Требуем сгенерировать
//конкретизацию шаблонной функции
FuncName<int>(ch) ;}
```



## Указатель на функцию

```
void err (char * p) // определение функции  
{...}
```

```
void (*p_func) (char *); // указатель на функцию  
...
```

```
Void main()
```

```
{p_func = &err; // получение адреса функции  
(*p_func)("data"); // вызов функции}
```

## Встроенные функции

```
inline int sum(int a, int b) {return (a+b);}
```