

Class, structure, union

➤ In C++ there is no fundamental difference between class and structure. The structure may contain not only data but also functions. The structure differs from class in that the default members of classes are private and structures are public. To declare a structure element as a private member, use the private keyword. Though a structure can contain methods, usually programmers use a structure to represent data and classes to represent objects.

Like a structure, a **union** may also be used to define a class. In C++, **unions** may contain both member functions and variables. They may also include constructors and destructors. **The most important being that all data elements share the same location in memory.** Union members are public by default.

Example 12.

- By difference from classes, unions cannot inherit from any classes or types. The union cannot be a base class.
- The members of the union cannot be
 - virtual functions
 - static variables
 - reference variables
 - object with an overloaded operator =

- Anonymous unions - do not contain the type name. Objects of the type defined by such a union cannot be declared. Anonymous unions tell the compiler that their member variables will be in the same memory area. Access to the components of the anonymous union is direct:

```
union  
{  
    int i;  
    char ch[4];  
}
```

```
i = 4;  
ch[0] = 'X';
```

- Anonymous unions cannot contain private members
- The names of the members of the anonymous union should not conflict with other variable names in the scope of the declaration.

Inline functions

- These are short functions that are not actually called. Instead, their body remains embedded in the program code where it is called.
- Such a technique makes it possible to bypass the function calling mechanism related to the procedure of creating a stack of formal arguments, passing the current argument values, returning values, etc. **Inline - function will be called much faster than a normal call.** This is an advantage of inline functions.
- The disadvantage of inline functions is that if the code of such inline functions is long and the inline functions are called too often, the program size will increase drastically.

Definition of inline functions:

```
#include <iostream>
using namespace std;

inline int even(int x)
{
    return (x%2);
}

void main()
{
    int retv = even(10); //retv = 0;
}
```

➤ **The inline function must be defined before it is first called.** (Otherwise, the compiler doesn't know which code to weave in).

➤ Depending on the compiler type (see compiler documentation), there may be a number of restrictions

- the function does not have to be recursive
- does not have to contain static variables
- does not have loops, *switch* or *goto* instructions

➤ **Functions - members of a class can also be inline.**

```
#include <iostream>
using namespace std;
```

```
class samp
{
    int i;
public:
    samp(int a);
    int get_i();
};
```

```
samp::samp(int a)
{
    i = a;
}
```

```
inline int samp::get_i()
{
    return i;
}
```

Default arguments of the function

In C++, it is allowed to assign certain default values to function parameters. If no such argument is given when calling the function, its value will be taken as the default value. This is the hidden form of function overload.

```

void fun(int i =0; int j=0);

int main()
{
    .....
    fun();          //OK, is passed i=0; j=0
    fun(5);         //OK, is passed i=5, j=0
    fun(24, 32);    //OK, is passed i=24, j=32
    return 0;
}

void fun(int i, int j)
{
    .....
}

```

It is impossible to pass the first parameter by default, and the second - not.

Default arguments must be given either in the prototype of the function or in the definition if the first call to this function goes after the definition. It is not allowed to enter both the default arguments in the prototype and in the definition.

All default parameters must be placed at the right relatively of parameters passed in the usual way.

```
void fun(int i, int j=0);    //OK.  
void fun(int i=0, int j);    //error: default parameter is at the left from parameter passing in usual way.
```

The default arguments must be constants or global parameters.

Default arguments and overloaded functions.

Example Square area $S = a * a$, the area of the rectangle $S = a * b$. Instead, to write overloaded functions of type

```
double get_area(double a, double b);    //area of rectangle  
double get_area (double a);             //area of square
```

is possible to write a single function:

```
double get_area (double a, double b = 0)  
{  
    if(b != 0.0)  
        return a*b;    // area of rectangle  
    else  
        return a*a;    // area of square  
}
```



```
int main()
{
    double aa = 10.0;
    cout << " area of square :" << get_area(aa) << endl;
    cout << " area of rectangle: " << get_area (aa, 20.0) << endl;
}
```

Passing default arguments to class constructors:

```
class myclass
{
    int i;
public:
    myclass(int ii = 0);
    .....
};
```

```

int main()
{
    //The constructor with a default argument is used to create both: initialized and
    //uninitialized objects.
    myclass ob(10);    //ob::i = 10
    myclass *ptr_ob;

    try
    {
        ptr_ob = new myclass [20]; //default constructor is required
    }
    catch(bad_alloc aa)
    {
        .....
    }
    .....
    delete [ ] ptr_ob;
    return 0;
}

```

Copy constructor with default arguments: you can create a copy constructor with additional arguments as long as all additional arguments are default arguments.

```

class myclass
{
    int i;
public:
    myclass(int ii = 0) : i(ii) { }
    myclass(const myclass &aa, double b = 0, char str[] = "Hello!");
    int get_i() { return i; }
    void set_i(int ii) { i = ii; }
};

myclass::myclass(const myclass &aa, double b, char *str)
{
    i = aa.get_i();
    cout << "copy constructor = " << i << " b = " << b << " str = " << str << endl;
}

int main()
{
    myclass tt(10);
    myclass tt1 = tt; //copy constructor is called
    .....
}

```

It should be remembered that the exaggeration in the use of the default arguments complicates the program readability.

You can use the default arguments in such case when we have a large program in which the function

```
void draw_rect(double x1, double y1, double x2, double y2);
```

After a few years, it turned out to be necessary to add the *int my_flag* parameter to the list of formal arguments. In order not to make changes to the old part of the code that does not use this flag (or maybe the sources of this part of the code are not available to us even) we modify so:

```
void draw_rect(double x1, double y1, double x2, double y2, int my_flag = 0);
```

We rewrite the *draw_rect* functions so that when *my_flag* = 0 it works the same as in the previous version. Then the old part of the code does not need any change, and the new code can be written using the new possibilities of the *draw_rect* function.

One more example of function overload:

```
int AfxMessageBox( LPCTSTR lpszText, UINT nType = MB_OK, UINT nIDHelp = 0 );
```

```
int AFXAPI AfxMessageBox( UINT nIDPrompt, UINT nType = MB_OK, UINT nIDHelp = (UINT ) -1 );
```

In WINDOWS: UINT – unsigned int; DWORD - unsigned long;

LPCTSTR lpszText – text array (const char *), which is displayed on dialog.

nIDPrompt – ID of resource (string table) – the same text array can be placed in resource (tables of the text arrays).

nType – type of buttons (OK; Yes, No; Yes, No, Cancel; Abort, Retry, Ignore; ..).

nIDHelp – help context (if exists).

Example: **resource**.

Overloaded functions and ambiguity - situations when the compiler is unable to choose which of the two (or more) overloaded functions to call. This will end up with a compilation error.

Most often, such a case arises with the automatic conversion of types:

```
float fun(float a)
{
    return a/20.0;
}

double fun(double a)
{
    return a/20.0;
}


int main()
{
    float    x = 40.0;
    double   y = 40.0;
    cout << fun(x) << endl;    //OK – float
    cout << fun(y) << endl;    //OK – double
    cout << fun(40) << endl;    // to which type to convert an integer argument? – error
    cout << fun((double)40) << endl;    // OK – explicit conversion is applied
}
```

The default arguments can cause ambiguity:

```
int fun(int ii)
{
    cout << ii << endl;
    return ii;
}

int fun(int ii, int jj = 0)
{
    cout << ii << jj << endl;
    return ii;
}

int main()
{
    int iii = 40, jjj = 60;
    cout << fun(iii, jjj) << endl; //OK,
    cout << fun(iii) << endl;      //error: which function should be called?
    .....
}
```



Calling the function by the pointer to function.

- Pass the address of the function as an argument to another function.

```
void qsort( void *base, size_t number, size_t width, int (* compare )(const void *, const void *) );
```

`int (* compare)(const void *, const void *)` – it is a declaration of the pointer to any function, which returns *int* type and has the both arguments of the type *const void **.

*void ** - the pointer of unspecified type; *const type * ptr* - preserve an object to which points the pointer *ptr* from any changes.

If we prepare a function

```
int my_comp(const void *e1, const void *e2)
{
    const int * a1 = (const int *)e1;
    const int * a2 = (const int *)e2;
    return (a1 < a2);    //ascending order
}
```

and pass its address to `qsort` `qsort((void *)arr, 10, sizeof(arr[0], my_comp);`
`qsort` function will use *my_comp* during sorting of array `int arr[10]` in ascending order.

- Dynamic choice which function should be called.

Example CallByPtr.

- Multithreading.

Example MultThread.

In C++, the declaration of a pointer to a function determines which of the overloaded functions will be called:

```
double funtt(double aa, double bb);
int funtt(int ii, int kk);

double (*ptr_fun_d)(double aa, double bb);
int (*ptr_fun_i)(int ii, int kk);

int main()
{
    ptr_fun_d = &funtt;
    ptr_fun_i = &funtt;
    cout << " double " << (*ptr_fun_d)(10.0, 10.5) << endl;
    cout << " int    " << (*ptr_fun_i)(15, 16) << endl;
    .....
}
```

```
double funtt(double aa, double bb)
{
    return bb - aa;
}
```

```
int funtt(int ii, int jj)
{
    return jj - ii;
}
```