

Object-oriented programming

based on the C language++
2th semester

- pointer this
- friend functions
- operators overloading

Pointer * This.

*Each object is accompanied by a pointer to itself - called a pointer **this** - this is an implicit argument in all references to elements within the object.*

Example 6. Using a pointer * this.

```
#include <iostream>
using namespace std;
class Test
{
public:
    Test (int = 0);
    void print() const;
private:
    int x;
};

Test::Test(int a) { x = a; }

void Test::print() const
{
    cout << "          x = " << x << endl;

    cout << "    this->x = " << this->x << endl;
    cout << " (*this).x = " << (*this).x << endl;
}

int main()
{
    Test a(12);
    a.print();
    return 0;
}
```

C:\

```
x = 12  
this->x = 12  
(*this).x = 12
```

```
Process returned 0 (0x0)   execution time : 0.141 s  
Press any key to continue.
```

Another example of the use of the index this:

```
Time & setHour (int);  
Time & setMinute (int);  
Time & setSecond (int);
```

```
Time & Time::setHour(int h)  
{  
    hour = (h >= 0 && h < 24) ? h : 0;  
    return *this;  
}
```

```
t.setHour(18).setMinute(30).setSecond(22);
```

- friendly functions

Friendly functions and classes

friendly features - a feature that is not a member of the class, but has access to class members declared in the fields **private** or **protected**.

friendly function declared within a class, the elements of which it needs access to the keyword friend.

Friendly feature can be an ordinary function or by another previously defined class.

One function can be friendly to multiple classes.

Friendly features - an example

```
class monstr;
```

```
class hero {
```

```
    public:
```

```
    void kill (monstr &);
```

```
};
```

```
class monstr{
```

```
    friend int steal_ammo(monstr &);
```

```
    friend void hero :: kill (monstr &);
```

```
};
```

```
    int steal_ammo(monstr & M) {return -M.ammo;}
```

```
void hero :: kill (monstr & M) {
```

```
    M.health = 0; M.ammo = 0;
```

```
}
```

Friendly classes - an example

```
class hero{  
    ...  
    friend class mistress;  
}  
class mistress {  
    ...  
    void f1 ();  
    void f2 ();  
}
```



- operations Overloading

2. Overloading operations

Operations are overloaded by compiling a description of the function (with the header and the body), the function name consists of the keyword **operator** and functions of the mark. For instance:

```
int operator < (const String &) const;
```

To use the operation on class objects, this operation must be overloaded, but there are two exceptions:

- 1) **Assignment (=) operation** It can be used with each class without explicit congestion. The default assignment operator is reduced to bit copy of class fields, but this bit copy is unacceptable for classes with dynamically created fields; for such classes to explicitly overload the assignment operator.

2)

operation of receipt address (K) It can be used with any class objects without overloading; it simply returns the address of an object in memory. But addressing operation can also overload.

Operations that can be overwhelmed:

+ - * /% ^ & |

~! = <> + = - = * =

/ = % = ^ = ^ = << >>

==! = <=> = && || ++ -

-> [] () new delete

Operations that can not be overloaded:

.. * ::? Sizeof

The recommended form of operator overloading.

Операция	Рекомендуемая форма перегрузки
Все унарные операции	Метод класса
= [] () ->	Обязательно метод класса
+= -= *= /= %= &= ^=	Метод класса
Остальные бинарные операции	Внешняя функция друг / метод класса
Поместить в поток << Взять из потока >>	Только внешняя функция друг

- under overload operations stored number of arguments, the priorities of operations and rules of association (from right to left or left to right), used in standard data types;
- for standard data types can not override the operation;
- function operations may not have default arguments;
- inherited functions-operations (excluding =);
- function operations may not be defined as static.

Format:

```
type operator operation (parameter list) {  
    body functions  
}
```

Function, operation can be determined:

- as a class method
- as a friendly class function
- as an ordinary function

Overloading unary operations

1. Within the class:

```
class monstr{  
    ...  
    monstr & Operator ++ ()  
    {++ health; return * this;}  
};  
  
monstr Vasia;  
cout << (++Vasia).get_health();
```

Overloading unary operations

2. How friendly features:

```
class monstr{  
    ...  
    friend      monstr & Operator ++ ( monstr & M);  
};  
monstr& Operator ++ (monstr & M) {++M.health; return M;}
```


3. Outside of Class:

```
void change_health (int he) {health = he;}  
...  
monstr & operator ++ (monstr & M) {  
    int h = M.get_health (); h ++;  
    M.change_health (h) ;  
    return M;}  
}
```


Overloading the postfix increment

```
class monstr{  
    ...  
    monstr operator ++ (int) {  
        monstrM (* this); health ++;  
        return M;  
    }  
};  
monstr Vasia;  
cout << (Vasia++) .get_health();
```

The presence of the argument (int) Indicates that it is post-increment



Overloading binary operators

1. Within the class:

```
class monstr {  
    ...  
    bool operator> (const monstr & M) {  
        if (health> M.get_health ())  
            return true;  
        return false; }  
};
```

2. Outside of class:

```
bool operator> (const monstr & M1, const monstr &  
M2) {  
    if (M1.get_health ()> M2.get_health ())  
        return true;  
    return false;  
}
```

assignment Overloading

Operation function must return a reference to the object for which it is due, and take as a parameter a single argument - a reference to the object is assigned

```
const monstr& Operator = (const monstr & M) {  
    // Check to see samoprisvaivanie:  
    if (& M == this) return * this;  
    if (name) delete [] name;  
    if (M.name) {name = new char [strlen(M.name) + 1];  
        strcpy(Name, M.name);}  
    else name = 0;  
    health = M.health; ammo =M.ammo; skin =M.skin;  
    return *this;}
```

```
monstr A (10), B, C;  
C = B = A;
```

operations Overloading new and delete

- they do not need to pass a parameter class type;

the first parameter of the function new and new [] be transmitted type of object size `size_t` (this is the type returned by the `sizeof` operator, defined in `<stddef.h>` header file); the call is passed to the function implicitly;

- they should be determined by the type of the return value `void *`, even return returns a pointer to other types of (usually a class);
- operation delete must have a return type `void` and the first argument of type `void *`;
- Operation allocation and deallocation are static class members.

```
class Obj {...};
```

```
class pObj{
```

```
...
```

```
private:
```

```
Obj * P;
```

```
};
```

```
pObj * P = new pObj;
```

```
static void * operator new (size_t size);
```

```
void operator delete (void * ObjToDie, size_t  
size);
```

```
#include <new.h>
```

```
SomeClass a = new (buffer) SomeClass(his_size);
```

Overload type driving operation

```
imya_novogo_tipa operator ();
```

```
monstr :: operator int () {  
    return health;  
}
```

```
...
```

```
monstr Vasia; cout << int (Vasia);
```

Overload operation function call

```
class if_greater {  
    public:  
    int operator () (Int a, int b) const {  
        return a > b;  
    }  
};
```

```
if_greater x;  
cout << x (1, 5) << endl; //x.operator () (1, 5))  
cout << if_greater () (5, 1) << endl;
```

indexing operation Overloading

```
class Vect{
public:
    explicit Vect(int n = 10);
    //initialization array:
    Vect(const int a [], int n);
    ~Vect() {Delete [] p; }
    int& Operator [] (int i);
    void Print ();
private:
    int* P;
    int size;
};
```


indexing operation Overloading

```
Vect::Vect(int n): size (n) {p = new int[Size];}
Vect::Vect(const int a [], int n): size (n) {
    p = new int[Size];
    for (int i = 0; i <Size; i++) p [i] = A [i]; }

int& Vect::operator [] (int i) {
    if (i <0 || i > = Size) {
        cout << "Incorrect index (i = "<< i << ")" <<
endl;
        cout << "completion programs"<< endl;
        exit (0); }
    return p [i];
}
```

indexing operation Overloading

```
void Vect:: Print () {  
    for (int i = 0; i <Size; i++) cout << p [i]  
    << "  
    cout << endl; }
```

```
int main () {  
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9,  
10};  
    Vect a (arr, 10);  
    a.Print();  
    cout << a [5] << endl;  
    cout << a [12] << endl;  
    return 0;  
}
```