ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

«BLACK BOX»

- Construction.
- Memory management.
- Overload operators.
- …

```cpp
struct Point {
    double x, y, z;

    Point() = default;

    Point(double _x, double _y, double _z)
        : x(_x), y(_y), z(_z) {}
};

Point a(10, 20, 30);
Point b(a);
Point c(Point(1, 2, 3));
Point d;
```

User defined constructor

Default copy constructor

Implicitly-declared move constructor

Default constructor

COPY CONSTRUCTOR

1. An object of the class is returned by value.

```
Point zeroPoint() {
    return Point(0, 0, 0);
}
```

2. An object of the class is passed (to a function) by value as an argument.

```
double norm(Point p) {
    return sqrt(p.x * p.x +
                p.y * p.y +
                p.z * p.z);
}
```

3. An object is constructed based on another object of the same class.

```
Point a(10, 20, 30);
Point b(a);

Point c = Point(4, 5, 6);
```

4. Exceptions

```
try {
    // ...
    throw std::runtime_error("Too bad!");
}

catch (std::runtime_error e) {
    // ...
}
```

COPY CONSTRUCTORS

```
X(const X& copy_from_me);
X(X& copy_from_me);
X(volatile X& copy_from_me);
X(const volatile X& copy_from_me);
X(X& copy_from_me, int = 0);
X(const X& copy_from_me, double = 1.0, int = 42);
```

```
struct Point {
    double x, y, z;

    Point(double _x, double _y, double _z)
        : x(_x), y(_y), z(_z) {}

    // This is bad code!
    Point(const Point &other) :
        x(other.x), y(other.y), z(other.z)
    {}
};
```

Such a constructor is automatically generated (shallow copy)

```cpp
struct Buffer {
    char *buf;
    size_t size;

    Buffer(size_t sz) :
        buf(new char[sz]), size(sz) {}

    ~Buffer() { delete buf; }
};
```

How it works?

```
Buffer::Buffer(const Buffer &other)
    : size(other.size)
{
    buf = new char[size];

    if (buf)
        memcpy(buf, other.buf, size);
}
```

Need user defined copy constructor!

```
Buffer fillBuffer() {
    Buffer b(1024);

    memset(b.buf, 0, b.size);

    return b;
}

Buffer mybuf = fillBuffer();
```

How many constructors will be called here?

MOVE CONSTRUCTOR

```
struct Buffer {
    char *buf;
    size_t size;

    Buffer(size_t sz);
    Buffer(const Buffer &other);
    Buffer(Buffer &&other) {
        buf = other.buf;
        size = other.size;
        other.buf = nullptr;
        other.size = 0;
    }

    ~Buffer();
};
```

```cpp
// Classic swap
template<class T>
void swap(T& a, T& b) {
    T tmp {a};                  // two copy of a
    a = b;                      // two copy of b
    b = tmp;                    // two copy of tmp
}
```

```cpp
// Almost perfect swap
template<class T>
void swap(T& a, T& b) {
    T tmp {static_cast<T&&>(a)};
    a = static_cast<T&&>(b);
    b = static_cast<T&&>(tmp);
}
```

```cpp
// Same code, but using std::move
template<class T>
void swap(T& a, T& b) {
    T tmp {std::move(a)};          // move from a
    a = std::move(b);              // move from b
    b = std::move(tmp);            // move from tmp
}
```

OVERLOADING THE ASSIGNMENT OPERATOR

- Implicitly-declared copy assignment operator (shallow copy).
- If there are pointers in a class, the default implementation may cause problems.

```cpp
struct Buffer {
    char *buf;
    size_t size;

    Buffer(size_t sz) :
        buf(new char[sz]), size(sz) {}

    ~Buffer() { delete buf; }
};

Buffer b1(100), b2(200);

b1 = b2; // Oops!
```

```
Buffer &Buffer::operator=(const Buffer &other)
{
    delete buf;

    buf = new char[size = other.size];

    if (buf)
        memcpy(buf, other.buf, size);

    return *this;
}
```

```
Buffer buf(100), buf2(200);

buf = buf;  // Oops!
```

```cpp
Buffer &Buffer::operator=(const Buffer &other)
{
    if (this != &other) {
        delete buf;

        buf = new char[size = other.size];
        if (buf)
            memcpy(buf, other.buf, size);
    }

    return *this;
}
```

Self-Assignment Protection

```cpp
Buffer &Buffer::operator=(Buffer &&other)
{
    buf = other.buf;
    buf.size = other.size;

    other.buf = nullptr;
    other.size = 0;

    return *this;
}
```

Move assignment operator

IMPLICIT MEMBERS

- X()
- X(const X &)
- X& operator=(const X &)
- X(X &&)
- X& operator=(X &&)
- ~X()

```
struct S {
    string a;
    int b;
};

S f(S arg) {
    S s0 {};   // default constructor: {"",0}
    S s1 {s0}; // copy constructor
    s1 = arg;  // copy assignment
    return s1; // move constructor
}
```

IMPLICIT MEMBERS

- **Default constructor**: if no other constructors.
- **Destructor**: if no destructor.
- **Copy constructor**: if no move constructor and no move assignment.
- **Copy assignment**: if no move constructor and no move assignment.
- **Move constructor**: if no destructor, no copy constructor and no copy nor move assignment.
- **Move assignment**: if no destructor, no copy constructor and no copy nor move assignment.

OPERATOR OVERLOADING

```
add(a, multiply(b, c));

// vs.

a + b * c;
```

- You cannot create new operators.
- The priority or associativity of operators cannot be affected.

Operator as member function

```
A a;
SomeType b;

// a + b

class A {
    // ...
    return_type operator+(SomeType b);
};

// b + a ?
```

Operator as non-member function

```
class A {
    // ...
    friend return_type operator+(const A &, SomeType);
    friend return_type operator+(SomeType, const A &);
};

inline return_type operator+(const A &a, SomeType b) {
    // ...
}


inline return_type operator+(SomeType b, const A &a) {
    // ...
}
```

ARITHMETIC

- **+   -   *   /   %**
- Both options can be used (as member and non-member function).
- **operator-** can be either binary or unary!

BITWISE OPERATORS

- **^ | & ~ << >>**
- Priority lower than arithmetic:
  a ^ n + b equivalently as **a ^ (n + b)**.
- Both options can be used (as member and non-member function).
- **<<** and **>>** stream insertion/extraction operators.

```cpp
ostream& operator<<(ostream& out, const Vector2D& vec) {//stream insertion
    out << "(" << vec.x() << ", " << vec.y() << ")";
    return out;
}

istream& operator>>(istream& in, Vector2D& vec) { // stream extraction
    double x, y;
    // ignore opening bracket
    in.ignore(1);

    // read x
    in >> x;
    vec.set_x(x);

    // ignore delimiter
    in.ignore(2);

    // read y
    in >> y;
    vec.set_y(y);

    // ignore closing bracket
    in.ignore(1);

    return in;
}
```

SIMPLE ASSIGNMENT OPERATOR

- As member function only!
- There is implicit implementation: shallow copy.
- If there are pointers in a class, the default implementation may cause problems.

COMPARISON OPERATORS

- **==   !=   <   <=   >   >=**
- Both options can be used (as member and non-member function).
- We can define all operators using two:
  - **#include <utility>**
  - **using namespace std::rel_ops;**
  - Define **operator==** and **operator<**
  - The remaining operators will provide STL

LOGICAL OPERATORS

- **! && ||**
- Builtin **operators &&** and **||** perform ***short-circuit*** evaluation, but overloaded operators behave like regular function calls and always evaluate both operands.

```cpp
bool Function1() { return false; }
bool Function2();

Function1() && Function2();

/////////////////////////////////

MyBool Function3() { return MyBool::FALSE; }
MyBool Function4();

bool operator&&(MyBool const &, MyBool const &);

Function3() && Function4();
```

Function2() will not be called,
but Function4() – will be called!

ASSIGNMENT OPERATORS

- **+=   -=   *=   /=   %=   &= |=   ^=   <<=   >>=**
- As member function only!
- Return ***this**.

INCREMENT/DECREMENT OPERATORS

```
SomeValue& SomeValue::operator++()              // prefix
{
    ++data;
    return *this;
}

SomeValue SomeValue::operator++(int unused)     // postfix
{
    SomeValue result = *this;
    ++data;
    return result;
}
```

```
SomeValue v;

++v;    // prefix ++
v++;    // postfix ++
```

Postfix operator implementation by prefix operator

```
// postfix
SomeValue SomeValue::operator++(int unused)
{
    SomeValue result = *this;
    ++(*this); // call SomeValue::operator++()
    return result;
}
```

SUBSCRIPT OR ARRAY INDEX OPERATOR [ ]

- As member function only!
- It takes only one argument of any type.
- Usually overloading two versions: <u>with</u> and <u>without</u> **const**.

```cpp
template <typename T>
class StupidVector {

    T array[100];
public:
    // ...
    T &operator[](size_t idx) {
        return array[idx];
    }

    const T &operator[](size_t idx) const {
        return array[idx];
    }
};
```

```cpp
template <typename T>
class Matrix;

template <typename T>
class MatrixColumn {
public:
    MatrixColumn(Matrix *m, size_t r)
        : matrix(m), row(r) {}

    T &operator[](int col) {
        return matrix->element(row, col);
    }

    Matrix *matrix;
    size_t row;
};

template <typename T>
class Matrix {
public:
    Matrix(int rows, int cols);

    // ...

    T &element(int row, int col);

    MatrixColumn<T> operator[](int row) {
        return MatrixColumn<T>(this, row);
    }
};
```

```cpp
Matrix<double> mat(3, 3);

mat[2][2] = 10;
```

FUNCTION CALL OPERATOR

- As member function only! No other restrictions.
- Actively used in STL to create functors.
    - **std::unary_function**
    - **std::binary_function**
    - …

```cpp
template <class _Arg, class _Result>
struct unary_function {
    typedef _Arg    argument_type;
    typedef _Result result_type;
};

template <class _Arg1, class _Arg2, class _Result>
struct binary_function {
    typedef _Arg1   first_argument_type;
    typedef _Arg2   second_argument_type;
    typedef _Result result_type;
};

template <class _Tp>
struct plus : public binary_function<_Tp, _Tp, _Tp> {
    _Tp operator()(const _Tp& __x, const _Tp& __y) const {
        return __x + __y;
    }
};

template <class _Tp>
struct negate : public unary_function<_Tp, _Tp> {
    _Tp operator()(const _Tp& __x) const {
        return -__x;
    }
};
```

```
#include <iostream>
#include <functional>
#include <algorithm>

using namespace std;

int main () {
    int numbers[] {10, -20, -30, 40, -50};
    int cx;
    cx = count_if(numbers, numbers+5, bind2nd(less<int>(), 0));
    cout << "There are " << cx << " negative elements.\n";
    return 0;
}
```

POINTER OPERATORS

- **operator&**
- **operator***
- **operator->**

```cpp
template <typename T>
class undeletable_pointer {
public:
    undeletable_pointer(T *ptr) : base(ptr) {}
    // ...
private:
    void operator delete(void *);
    T *base;
};

struct SomeObject {
    typedef undeletable_pointer<SomeObject> undeletable_ptr;

    undeletable_ptr operator&() { return this; }
};
```

operator->

- Return a pointer.

`a->b` interpreted as

```
(a.operator->())->b;
```

smart pointer

```
class Err {};

class Giant {};

class Big {
public:
    Big() { throw Err(); }
};

class MyClass {
    Giant *giant;
    Big   *big;
public:
    MyClass(): giant(new Giant()), big(new Big()) {}

    ~MyClass() { delete giant; delete big; }
};

int main()
{
    try {
        MyClass myobject;
    } catch (Err) {}

    return 0;
}
```

```cpp
template <typename T>
class SmartPtr {
    T *ptr;
public:
    SmartPtr(T *p) : ptr(p) {};
    T& operator*()  { return *ptr; }
    T* operator->() { return ptr; }

    ~SmartPtr() {
        delete ptr;
    }
};
```

```
template <typename T>
class SmartPtr {
    T *ptr;
public:
    explicit SmartPtr(T *p = nullptr) : ptr(p) {}
    T& operator*()  const { return *ptr; }
    T* operator->() const { return ptr;  }

    SmartPtr(SmartPtr<T> &other) : ptr(other.release()) {}

    SmartPtr operator=(SmartPtr<T>& other) {
        if (this != &other)
            reset(other.release());
        return *this;
    }

    ~SmartPtr() { delete ptr; }

    T *release() {
       T *oldPtr = ptr;
       ptr = nullptr;
       return oldPtr;
    }

    void reset(T *newPtr) {
       if (ptr != newPtr) {
           delete ptr;
           ptr = newPtr;
       }
    }
};
```

TYPE CONVERSION

```
class Y {
    // ...
};

ostream &operator<<(ostream &os, const Y &y);

class X {
    // ...
    operator bool() const;
    operator Y() const;
};

X x;
if (x) {
    // ...
}

Y y(x);
cout << x;
```

Useful Implicit Conversion

```
class complex {
    double re, im;
public:
    complex(double r = 0, double i = 0)
      : re(r), im(i) {}
    // ...
};

bool operator==(complex, complex);

void f(complex x, complex y) {
    x==y;     // operator==(x,y)
    x==3;     // operator==(x,complex(3))
    3==y;     // operator==(complex(3),y)
}
```

Useless Implicit Conversion

```
class Date {
    int d, m, y;
public:
    Date(int dd = today.d, int mm = today.m, int yy = today.y);
    // ...
};

void my_fct(Date d);

void f() {
    Date d {15};    // 15th of this month
    // ...
    my_fct(15);     // Oops.
    d = 15;         // Hmm...
    // ...
}
```

- If a class has a constructor with one argument, the compiler can do "type conversion" by creating a temporary object.
- The **explicit** specifier prevents the compiler converting types.

```cpp
void f(const Y &);

class Y {
    Y(const X &);
};

X x;
f(x);

//////////////////

class Array {
public:
    Array(int size);
};

Array a('?');
```

```cpp
void f(const Y &);

class Y {
    explicit Y(const X &);
};

X x;
f(x);  // Error

//////////////////

class Array {
public:
    explicit Array(int size);
};

Array a('?');   // Error
```

explicit

```
struct A {
    // implicit conversion to int
    operator int() const { return 100; }

    // explicit conversion to std::string
    explicit operator std::string() const { return "explicit"; }
};

int main() {
    A a;
    int i = a;         // OK - implicit conversion
    std::string s = a; // Error. Need explicit conversion
    std::string t = static_cast<std::string>(a); // OK
}
```

Delete implicit operators

```
class X {
public:
    // ...
    void operator=(const X&) = delete;
    void operator&() = delete;
    void operator,(const X&) = delete;
    // ...
};

void f(X a, X b) {
    a = b;      // Oh!
    &a;         // Oh! Oh!
    a,b;        // Oh! Oh! Oh!
}
```

OPERATORS AND NAMESPACE

Binary operator **x@y**, where **x** is type X, and **y** is type **Y:**

- If X is a class, look for the **operator@** as a member of X or its base class.

- Search **operator@** for expression **x@y**.

- If X is declared in the namespace **N**, look for **operator@** in that namespace.

- If **Y** is declared in the namespace **M**, look for **operator@** in that namespace.

OPERATORS THAT CANNOT BE OVERLOADED

- **? :** (ternary)
- . (member access or dot operator)
- .* (pointer to member operator)
- :: (namespace)
- **sizeof**
- **alignof**
- **typeid**