OOP

Gavrilut Dragos

Course 4

Summary

- Destructor
- C/C++ operators
- Operators for classes
- Operations with Objects

- Destructor

Destructor

- A destructor function is called whenever we want to free the memory that an object occupies.
- A destructor (if exists) is only one and has no parameters.
- A destructor can not be static
- A destructor can have different access modifiers (public/private/protected).

App.cpp

```
class Date
{
private:
	int x;
public:
	Date();
	~Date();
};
Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
	Date d;
}
```

Destructor

- The most common usage of the destructor is to deallocate the memory that has been allocated within the constructor or other functions.

App.cpp

```
class String
{
    char * text;
public:
    String(const char * s)
    {
        text = new char[strlen(s) + 1];
        memcpy(text, s, strlen(s) + 1);
    }
    ~String()
    {
        delete text;
        text = nullptr;
    }
};

void main()
{
    String * s = new String("C++");
    // some operations
    delete s;
}
```

Destructor

- This code will not compile. The destructor of class Date is private.

App.cpp

```
class Date
{
private:
      int x;
public:
      Date();
private:
      ~Date();
};
Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
      Date d;
}
```

error C2248: 'Date::~Date': cannot access private member declared in class 'Date'
note: compiler has generated 'Date::~Date' here
note: see declaration of 'Date'

Destructor

- This code will compile – because the destructor will (even if private) is not be called (the object is created in the heap memory and it is never deallocated).

App.cpp

```
class Date
{
private:
	int x;
public:
	Date();
private:
	~Date();
};
Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
	Date *d = new Date();
}
```

Destructor

- This code will not compile because “***delete d***” call will attempt to use a private destructor.

App.cpp

```
class Date
{
private:
      int x;
public:
      Date();
private:
      ~Date();
};
Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
      Date *d = new Date();
      delete d;
}
```

error C2248: 'Date::~Date': cannot access private member declared in class 'Date'
note: compiler has generated 'Date::~Date' here
note: see declaration of 'Date''

Destructor

- This code will compile. The destructor is private , but it can be access by a method from its class (In this case DestroyData).

App.cpp

```
class Date
{
private:
      int x;
public:
      Date();
      static void DestroyData(Date *d);
private:
      ~Date();
};
Date::Date() : x(100) { ... }
Date::~Date() { ... }
void Date::DestroyData(Date *d)
{
     delete d;
}
void main()
{
      Date *d = new Date();
      Date::DestroyData(d);
}
```

Destructor

Let's consider the following class:

App.cpp

```
class Date {
public:
    ~Date() { printf("dtor was called \n"); }
};
```

The destructor is called when:

A. When program ends, for every global variable

```
Date d;
int main() { return 0; }
```

B. When a function/method ends for every local variable

```
int main() {
   Date d;
   return 0;
}
```

C. When the execution exists a scope (for variable defined within a specific scope)

```
int main() {
   for (int tr=0;tr<10;tr++) {
      Date d;
   }
   return 0;
}
```

D. When the ***delete*** operator is called over a heap allocated instance

```
int main() {
   Date *d = new Date();
   delete d;
   return 0;
}
```

Destructor

- Objects are destroyed in the reverse order of their creation (similar to the way a stack works → first created is the last destroyed).

App.cpp

```
class Tree {
public:
    ~Tree() { printf("dtor: Tree\n"); }
};
class Car {
public:
    ~Car() { printf("dtor: Car\n"); }
};
class Animal {
public:
    ~Animal() { printf("dtor: Animal\n"); }
};
class Date
{
    Tree t;
    Car c;
    Animal a;
public:
    ~Date() { printf("dtor: Date\n"); }
};
void main()
{
    Date d;
}
```

Outputs:
dtor: Date
dtor: Animal
dtor: Car
dtor: Tree

Destructor

- Objects are destroyed in the reverse order of their creation (similar to the way a stack works → first created is the last destroyed).

App.cpp

```
class Tree {
public:
    ~Tree() { printf("dtor: Tree\n"); }
};
class Car {
public:
    ~Car() { printf("dtor: Car\n"); }
};
class Animal {
public:
    ~Animal() { printf("dtor: Animal\n"); }
};
class Date
{
    Tree t;
    Car c;
    Animal a;
public:
    ~Date() { printf("dtor: Date\n"); }
};
void main()
{
    Date d;
}
```

Outputs:
dtor: Date
dtor: Animal

```
mov     dword ptr [this],ecx
push    offset string "dtor: Date\n"
call    _printf
add     esp,4

mov     ecx,dword ptr [this]
add     ecx,2
call    Animal::~Animal

mov     ecx,dword ptr [this]
add     ecx,1
call    Car::~Car

mov     ecx,dword ptr [this]
call    Tree::~Tree
```

Destructor

- If the destructor is missing, but the class has data members that have their own destructors, one will be created by default !

App.cpp

```
class Tree {
public:
    ~Tree() { printf("dtor: Tree\n"); }
};
class Car {
public:
    ~Car() { printf("dtor: Car\n"); }
};
class Animal {
public:
    ~Animal() { printf("dtor: Animal\n"); }
};
class Date
{
    Tree t;
    Car c;
    Animal a;
public:
};
void main()
{
    Date d;
}
```

Outputs:
dtor: Animal
dtor: Car
dtor: Tree

Destructor

- Let's analyze the following code. Each object created has its unique ID. Upon execution the following code will output:

App.cpp

```
int global_id = 0;
class Date
{
    int id;
public:
    Date() { global_id++; id = global_id; printf("ctor id:%d\n", id); }
    ~Date() { printf("dtor id:%d\n", id); }
};
void main()
{
    Date *d = new Date();
    delete d;
}
```

Outputs:
ctor id: 1
dtor id: 1

Destructor

- Let’s analyze the following code. Each object created has its unique ID. Upon execution the following code will output:

App.cpp

```
int global_id = 0;
class Date
{
    int id;
public:
    Date() { global_id++; id = global_id; printf("ctor id:%d\n", id); }
    ~Date() { printf("dtor id:%d\n", id); }
};
void main()
{
    Date *d = new Date[5];
    delete d;
}
```

An array of 5 instances of type Data is created.

Outputs:
ctor id: 1
ctor id: 2
ctor id: 3
ctor id: 4
ctor id: 5
dtor id: 1

- This program will crash as only the first object in the “d” array is destroyed. And it is not in the right order anyway.

Destructor

- Let’s analyze the following code. Each object created has its unique ID. Upon execution the following code will output:

App.cpp

```
int global_id = 0;
class Date
{
    int id;
public:
    Date() { global_id++; id = global_id; printf("ctor id:%d\n", id); }
    ~Date() { printf("dtor id:%d\n", id); }
};
void main()
{
    Date *d = new Date[5];
    delete [] d;
}
```

Outputs:
ctor id: 1
ctor id: 2
ctor id: 3
ctor id: 4
ctor id: 5
dtor id: 5
dtor id: 4
dtor id: 3
dtor id: 2
dtor id: 1

- Now the program runs correctly.
- Whenever an array of instances is created into the heap, use ***delete[]*** operator to destroy it and not ***delete*** operator.
- ***delete[]*** operator will call the destructor function (if any) for every object in the array in the reverse order (starting from the last and moving forward to the first).

- C/C++ operators

Operators

- Depending on that operator's necessary number of parameters there are:
  - Unary
  - Binary
  - Ternary
  - Multi parameter
- Depending on the operation type, there are:
  - Arithmetic
  - Relational
  - Logical
  - Bitwise operators
  - Assignment
  - Others
- Depending on the overloading possibility
  - Those that can be overloaded
  - Those that can NOT be overloaded

Arithmetic operators

Operator	Type	Overload	Format	Returns
+	Binary	Yes	A + B	Value/reference
-	Binary	Yes	A – B	Value/reference
*	Binary	Yes	A * B	Value/reference
/	Binary	Yes	A / B	Value/reference
%	Binary	Yes	A % B	Value/reference
++ (post/pre-fix)	Unary	Yes	A++ or ++A	Value/reference
-- (post/pre-fix)	Unary	Yes	A-- or --A	Value/reference

Relational operators

Operator	Type	Overload	Format	Returns
==	Binary	Yes	A == B	bool or Value/reference
>	Binary	Yes	A > B	bool or Value/reference
<	Binary	Yes	A < B	bool or Value/reference
<=	Binary	Yes	A <= B	bool or Value/reference
>=	Binary	Yes	A >= B	bool or Value/reference
!=	Binary	Yes	A != B	bool or Value/reference

Logical operators

Operator	Type	Overload	Format	Returns
&&	Binary	Yes	A && B	bool or Value/reference
||	Binary	Yes	A || B	bool or Value/reference
!	Unary	Yes	!	bool or Value/reference

Bitwise operators

Operator	Type	Overload	Format	Returns
&	Binary	Yes	A & B	Value/reference
|	Binary	Yes	A | B	Value/reference
^	Binary	Yes	A ^ B	Value/reference
<<	Binary	Yes	A << B	Value/reference
>>	Binary	Yes	A >> B	Value/reference
~	Unary	Yes	~A	Value/reference

Assignment operators

Operator	Type	Overload	Format	Returns
=	Binary	Yes	A = B	Value/reference
+=	Binary	Yes	A += B	Value/reference
-=	Binary	Yes	A -= B	Value/reference
*=	Binary	Yes	A *= B	Value/reference
/=	Binary	Yes	A /= B	Value/reference
%=	Binary	Yes	A %= B	Value/reference
>>=	Binary	Yes	A >>= B	Value/reference
<<=	Binary	Yes	A <<= B	Value/reference
&=	Binary	Yes	A &= B	Value/reference
^=	Binary	Yes	A ^= B	Value/reference
|=	Binary	Yes	A |= B	Value/reference

Operators (others)

Operator	Type	Overload	Format	Returns
sizeof	Unary	No	sizeof(A)	Value
new	Unary	Yes	new A	pointer (A*)
delete	Unary	Yes	delete A	<None>
Condition (?)	Ternary	No	C ? A:B	A or B depending on the evaluation of C
:: (scope)		No	A::B	
Cast (type)	Binary	Yes	(A)B or A(B)	B casted to A
-> (pointer)	Binary	Yes	A->B	B from A
. (member)	Binary	Yes	A.B	B from A
[] (index)	Binary	Yes	A[B]	Value/reference
() (function)	Multi	Yes	A(B,C,…)	Value/reference
, (list)	Binary	Yes	(A,B)	Val/ref for (A follow by B)

Operators (evaluation order)

1. :: (scope)
2. () [] -> . ++ - -
3. + - ! ~ ++ - - (type)* & sizeof
4. * / %
5. + -
6. << >>
7. < <= > >=
8. == !=
9. &
10. ^
11. |
12. &&
13. ||
14. ?:
15. = += -= *= /= %=>>= <<= &= ^= |=
16. ,

Operators for classes

Operators for classes

- A class can define a series of special functions that behave exactly as an operator – that is to allow the program to explain how the compiler should understand certain operations between classes
- Use keyword: "***operator***"
- These functions can have various access operators (and they comply to rules imposed by the operators – if an operator is declared private then it can only be accessed within the class)
- Operators can be implemented outside the classes – in this case, if it is needed, they can be declared as "***friend***" functions in order to be accessed by private members from a class

Operators for classes

- In this case the ***operator+*** is overloaded allowing addition operation between an *Integer* and another *Integer*

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (const Integer &i);
};
int Integer::operator+(const Integer &i)
{
	return value + i.value;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int x = n1 + n2;
}
```

Operators for classes

- In this case the ***operator+*** is overloaded allowing addition operation between an *Integer* and another *Integer*
- The addition operation is applied for the left parameter, the right parameter being the argument. In other words: “***n1+n2***” ⇔ “***n1.operator+(n2)***”

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (const Integer &i);
};
int Integer::operator+(const Integer &i)
{
	return value + i.value;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int x = n1 + n2;
}
```

Operators for classes

- Parameters don’t have to be a const or a reference. Using the operator is similar to using a function (all of the promotion rules apply).
- It is however recommended to use const references when the result of an operator does not modify the arguments

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (Integer &i);
};
int Integer::operator+(Integer &i)
{
	return value + i.value;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int x = n1 + n2;
}
```

Operators for classes

- Similarly, the return value does not have a predefine type (e.g. while the usual understanding is that adding, multiplying, etc of two values of the same type will produce a result of the same type, this is not mandatory).

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer operator+ (Integer i);
};

Integer Integer::operator+(Integer i)
{
	Integer res(value+i.value);
	return res;
}

void main()
{
	Integer n1(100);
	Integer n2(200);
	Integer n3(0);
	n3 = n1 + n2;
}
```

Operators for classes

- Operators work as a function. They also can be overloaded.
- In this case the Integer class supports an addition operation between two Integer objects, or between an Integer object and a float variable

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (const Integer &i) { ... };
	int operator+(float nr);
};
int Integer::operator+(float nr)
{
	return value + (int)nr;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int x = n1 + n2;
	int y = n1 + 1.2f;
}
```

Operators for classes

- Operators work as a function. They also can be overloaded.
- In this case the code does not compile because there already is an “***operator+***” function with a “float” parameter

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (const Integer &i) { ... };
	int operator+(float nr);
	float operator+(float nr);
};
int Integer::operator+(float nr)
{
	return value + (int)nr;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int x = n1 + n2;
	int y = n1 + 1.2f;
}
```

Operators for classes

- Pay attention at operators' usage order and don’t assume bijection. In this case the code does NOT compile. The Integer class handles the addition between an Integer and a float, but not the other way around (between a float and an Integer). This is not possible with a function from the class.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (const Integer &i) { ... };
	int operator+(float nr);
};
int Integer::operator+(float nr)
{
	return value + (int)nr;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int x = n1 + n2;
	int y = 1.2f + n1;
}
```

error C2677: binary '+': no global operator found which takes type 'Integer' (or there is no acceptable conversion)

Operators for classes

- The code compiles – friend functions solve both cases (*Integer+float* and *float+Integer*)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend int operator+ (const Integer &i, float val);
	friend int operator+ (float val, const Integer &i);
};
int operator+(const Integer &i, float val)
{
	return i.value + (int)val;
}
int operator+(float val, const Integer &i)
{
	return i.value + (int)val;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int y = (1.2f+n1)+(n2+1.5f);
}
```

Operators for classes

- The code compiles – friend functions solve both cases (*Integer+float* and *float+Integer*)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend int operator+ (const Integer &i, float val);
	friend int operator+ (float val, const Integer &i);
};
int operator+(const Integer &i, float val)
{
	return i.value + (int)val;
}
int operator+(float val, const Integer &i)
{
	return i.value + (int)val;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	int y = (1.2f+n1) + (n2+1.5f);
}
```

Operators for classes

- This code will NOT compile. There are two operators defined (one as part of the class, and the other one as a friend function), both of them referring to the same operation (***Integer*** + ***Integer***).

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator+ (Integer i);
	friend int operator+ (Integer n1, Integer n2);
};
int Integer::operator+(Integer i)
{
	return this->value + i.value;
}
int operator+ (Integer n1, Integer n2)
{
	return n1.value + n2.value;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	Integer n3(0);
	n3 = n1 + n2;
}
```

```
error C2593: 'operator +' is ambiguous
note: could be 'int Integer::operator +(Integer)'
note: or       'int operator +(Integer,Integer)'
```

Operators for classes

- Relational operators are defined exactly as the arithmetic ones. From the compiler point of view, there is no real difference between those two.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator > (const Integer & i);
};
bool Integer::operator > (const Integer & i)
{
	if (value > i.value)
		return true;
	return false;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	if (n2 > n1)
		printf("n2 mai mare ca n1");

}
```

Operators for classes

- Relational operators do not need to return a bool even though this is what is expected from them. Keep in mind that the compiler does not differentiate between arithmetic or logical operator. In this case, the ***operator>*** returns an object.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer operator > (const Integer & i);
	void PrintValue();
};
void Integer::PrintValue()
{
	printf("Value is %d", value);
}
Integer Integer::operator > (const Integer & i)
{
	Integer res(this->value + i.value);
	return res;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	(n1 > n2).PrintValue();
}
```

Operators for classes

- The same logic applies for logical operators as well (from the compiler point of view they are not different from the arithmetic or relational operators).

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer operator && (const Integer & i)
	void PrintValue();
};
void Integer::PrintValue()
{
	printf("Value is %d", value);
}
Integer Integer::operator && (const Integer & i)
{
	Integer res(this->value + i.value);
	return res;
}
void main()
{
	Integer n1(100);
	Integer n2(200);
	(n1 && n2).PrintValue();
}
```

Operators for classes

- An unary operator does not have a parameter (if it is defined within the class) or ***one*** parameter if it is defined as a “friend” function.
- Similar to the binary operators, there is no restriction for what these methods return. In the case below x will have the value 80.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator ! ();
};
int Integer::operator ! ()
{
	return 100 - this->value;
}
void main()
{
	Integer n1(20);
	int x = !n1;
}
```

Operators for classes

- The presented methods can be applied in the same way for the following operators:

<table>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>></td><td><</td><td>>=</td><td><=</td><td>!=</td></tr>
<tr><td>==</td><td>&</td><td>|</td><td>&&</td><td>||</td><td>^</td><td>!</td><td>~</td><td></td><td></td></tr>
</table>

- For these cases, it is recommended to use ***friend*** functions and not to create methods o within the class
- It is indicated, as much as possible, to add such functions with parameter combinations (class with int, int with class, class with double, double with class, etc)
- The operators can also return objects and/or references to an object. In these cases that object is then further used in the evaluation of the expression of which it is a part of.

Operators for classes

- In case of assignment, it is recommended to return a reference to the object that gets a value assign to. This will allow that reference to be further used in other expression.
- There is a special assignment operator called ***move assignment*** that can be used with a parameter that is a temporary reference ("&&")

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer& operator = (int val);
};
Integer& Integer::operator = (int val)
{
	value = val;
	return (*this);
}
void main()
{
	Integer n1(20);
	n1 = 20;
}
```

Operators for classes

- However, it is NOT mandatory to return a reference. The code below returns a bool value.
- After the execution of the code, ***n1.value*** will be 30, and ***res*** will be true

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator = (int val);
};
bool Integer::operator = (int val)
{
	value = val;
	return (val % 2) == 0;
}
void main()
{
	Integer n1(20);
	bool res = (n1 = 30);
}
```

Operators for classes

- Some operators (operator=, operator[], operator(), operator->) can not be a static function (be used outside the class through **friend** specifier).
- This case will not compile.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend bool operator= (Integer &i, int val) { i.value = val; return false; };
};

void main()
{
	Integer n1(20);
	bool res = (n1 = 30);
}
```

error C2801: 'operator =' must be a non-static member

Operators for classes

- However, the rest of assignment operators (+=, -=, *=, etc) can be implemented in this way.
- In this case the code compiles, res will have the value true and ***n1.value*** will be 30

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend bool operator += (Integer & i, int val);
};
bool operator += (Integer &i, int val)
{
	i.value = val;
	return true;
}
void main()
{
	Integer n1(20);
	bool res = (n1 += 30);
}
```

Operators for classes

- Be careful when using references and when a value. In this case the "***operator+***" is called, but with a copy of the class Integer. As a result, the value of ***n1.value*** will NOT change (will remain 20).

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend bool operator += (Integer i, int val);
};
bool operator += (Integer i, int val)
{
	i.value = val;
	return true;
}
void main()
{
	Integer n1(20);
	bool res = (n1 += 30);
}
```

Operators for classes

- Since **friend** functions are allowed, the order of the parameters can be changed. In this case, the code compiles even if “***30 &= …***“ does not make any sense.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend bool operator &= (int val, Integer i);
};
bool operator &= (int val, Integer i)
{
	i.value = val;
	return true;
}
void main()
{
	Integer n1(20);
	bool res = (30 &= n1);
}
```

Operators for classes

- A different case refers to postfix/prefix operators (++ and --)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator++ ();
	bool operator++ (int value);
};
bool Integer::operator++ ()
{
	value++;
	return true;
}
bool Integer::operator++ (int val)
{
	value += 2;
	return false;
}
void main()
{
	Integer n1(20);
	bool res = (n1++);
}
```

Operators for classes

- In this case, the postfix form is being executed (n1.value = 22, res = false)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator++ ();
	bool operator++ (int value);
};
bool Integer::operator++ ()
{
	value++;
	return true;
}
bool Integer::operator++ (int val)
{
	value += 2;
	return false;
}
void main()
{
	Integer n1(20);
	bool res = (n1++);
}
```

val=0, **must** be int type

Operators for classes

- In this case the prefix form is being executed (n1.value = 21, res = true)

App.cpp

```
class Integer
{
      int value;
public:
      Integer(int val) : value(val) {}
      bool operator++ ();
      bool operator++ (int value);
};
bool Integer::operator++ ()
{
      value++;
      return true;
}
bool Integer::operator++ (int val)
{
      value += 2;
      return false;
}
void main()
{
      Integer n1(20);
      bool res = (++n1);
}
```

Operators for classes

- Prefix/postfix operators can be "***friend***" functions. Normally the first parameter of the friend function has to be a reference type. After execution n1.value = 22, res = false

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend bool operator++ (Integer &i);
	friend bool operator++ (Integer &i,int value);
};
bool operator++ (Integer &i)
{
	i.value++;
	return true;
}
bool operator++ (Integer &i,int val)
{
	i.value += 2;
	return false;
}
void main()
{
	Integer n1(20);
	bool res = (n1++);
}
```

Operators for classes

- Postfix/prefix operators have a special meaning
  - PostFix – the value is returned first and then the operation is executed
  - Prefix – the operation is executed first and then the value is returned

App.cpp

```
void main()
{
	int x = 3;
	int y;
	y = x++;
	int z;
	z = ++x;

}
```

```
mov	eax,dword ptr [x]
mov	dword ptr [y],eax
mov	ecx,dword ptr [x]
add	ecx,1
mov	dword ptr [x],ecx
```

- In the first case y takes x's value and then the increment operation for x is being done. Meaning that y will be equal with 3 and x with 4.

Operators for classes

- Postfix/prefix operators have a special meaning
  - PostFix – the value is returned first and then the operation is executed
  - Prefix – the operation is executed first and then the value is returned

App.cpp

```
void main()
{
	int x = 3;
	int y;
	y = x++;
	int z;
	z = ++x;

}
```

- In the first case y takes x's value and then the increment operation for x is being done. Meaning that y will be equal with 3 and x with 4.
- In the second case, first the increment operation for x is done and then the assignation towards z, meaning that z will be equal with 5 and x also with

```
mov	eax,dword ptr [x]
add	eax,1
mov	dword ptr [x],eax
mov	ecx,dword ptr [x]
mov	dword ptr [z],ecx
```

Operators for classes

- Prefix/postfix operators can be modified to have the desired behavior (postfix, prefix) in the following way:

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer& operator++ ();
	Integer  operator++ (int value);
};
Integer& Integer::operator++ ()
{
	value += 1;
	return (*this);
}
Integer Integer::operator++ (int)
{
	Integer tempObject(value);
	value += 1;
	return (tempObject);
}
void main()
{
	Integer n1(20);
	n1++;
}
```

Operators for classes

- Prefix/postfix operators can be modified to have the desired behavior (postfix, prefix) in the following way:

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer& operator++ ();
	Integer  operator++ (int value);
};
Integer& Integer::operator++ ()
{
	value += 1;
	return (*this);
}
Integer Integer::operator++ (int)
{
	Integer tempObject(value);
	value += 1;
	return (tempObject);
}
void main()
{
	Integer n1(20);
	n1++;
}
```

Operators for classes

- Prefix/postfix operators can be modified to have the desired behavior (postfix, prefix) in the following way:

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer& operator++ ();
	Integer  operator++ (int value);
};
Integer& Integer::operator++ ()
{
	value += 1;
	return (*this);
}
Integer Integer::operator++ (int)
{
	Integer tempObject(value);
	value += 1;
	return (tempObject);
}
void main()
{
	Integer n1(20);
	++n1;
}
```

Operators for classes

- A special operator is **new**. **new** has a special format (it has to return ***void****** and has a ***size***_t first parameter).
- The new operator cannot be used as a friend function.
- The ***size***_t parameter represents the size of the object to be allocated.
- The new operator does not call the constructor, it is expected to allocate memory for the current object. In this case, after execution ***GlobalValue = 100***. The constructor (if any) will be called automatically by the compiler after the memory has been allocated.

App.cpp

```
int GlobalValue = 0;
class Integer {
	int value;
public:
	Integer(int val) : value(val) {}
	void* operator new(size_t t);
};
void* Integer::operator new (size_t t) {
	return &GlobalValue;
}
void main() {
	Integer *n1 = new Integer(100);
}
```

Operators for classes

- If ***new operator*** is declared with multiple parameters, they can be called/used in the following way:

App.cpp

```
int GlobalValue = 0;

class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	void* operator new(size_t t,int value);
};
void* Integer::operator new (size_t t,int value)
{
	return &GlobalValue;
}
void main()
{
	Integer *n1 = new(100) Integer(123);
}
```

- The functions/methods that overload ***new*** with multiple parameters are also called ***placement new***

Operators for classes

- The new [] operator has a similar behavior. It is used for allocating multiple objects. It has a similar format: it must return a ***void**** and the first parameter is also a ***size**_t (that represent the amount of memory needed for all of the elements in the vector).
- For the following example to work, a default constructor is required. After the execution, all elements from GlobalValue have their value equal to 1.

App.cpp

```
int GlobalValue[100];
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer() : value(1) {}
	void* operator new [](size_t t);
};
void* Integer::operator new[] (size_t t) { return &GlobalValue[0]; }
void main()
{
	Integer *n1 = new Integer[100];
}
```

Operators for classes

- ***new[]*** operator can also have several parameters. The following example shows an example on how such construct can be used.

App.cpp

```
int GlobalValue[100];
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	Integer() : value(1) {}
	void* operator new[] (size_t t,int value);
};
void* Integer::operator new[] (size_t t,int value)
{
	return &GlobalValue;
}
void main()
{
	Integer *n1 = new(123) Integer[100];
}
```

Operators for classes

- Generally speaking, the normal behavior for operators that assure the allocation is the following:

Operator
void* operator new (size_t size)
void* operator new[] (size_t size)
void operator delete (void* object)
void operator delete[] (void* objects)

- It is recommended for the new and delete operators to throw exceptions

Operators for classes

- Another special operator is the **cast** operator
- This operator allows the transformation of an object from one type to another
- Being a casting operator, we do not have to provide the return type (it is considered the type we are casting to) → in the next example: ***float***

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	operator float();
};
Integer::operator float()
{
	return float(value * 2);
}
void main()
{
	Integer n1(2);
	float f = (float)n1;
}
```

Operators for classes

- Cast operators are also use when such a conversion is explicitly required.
- As in the previous case, the value for f will be 4.0
- Cast operators cannot be used with **friend** specifier

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	operator float();
};
Integer::operator float()
{
	return float(value * 2);
}
void main()
{
	Integer n1(2);
	float f = n1;
}
```

Operators for classes

- Make sure to pay attention to all operators. In this case : f = 4.2

App.cpp

```
class Integer
{
      int value;
public:
      Integer(int val) : value(val) {}
      operator float();
};
Integer::operator float()
{
      return float(value * 2);
}
void main()
{
      Integer n1(2);
      float f = n1 + 0.2f;
}
```

Operators for classes

- However, in this case f = 20.2 due to the addition operator (**operator+**)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	operator float();
	float operator + (float f);
};
Integer::operator float()
{
	return float(value * 2);
}
float Integer::operator+ (float f)
{
	return value * 10.0f + f;
}
void main()
{
	Integer n1(2);
	float f = n1+0.2f;
}
```

Operators for classes

- The indexing operators allow the usage of [] for a certain object.
- They have only one restriction and that is that they only have one parameter - but this parameter can be anything and the return value also can be of any kind. Also, the indexing operator cannot be a friend function/ outside the current object
- In this case, ret=***true*** because byte 1 from ***n1.value*** is set

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator [](int index);
};
bool Integer::operator [](int index)
{
	return (value & (1 << index)) != 0;
}
void main()
{
	Integer n1(2);
	bool ret = n1[1];
}
```

Operators for classes

- The following example uses a different key (a ***const char ****) for the index operator []/

App.cpp

```
class Integer
{
      int value;
public:
      Integer(int val) : value(val) {}
      bool operator [](const char *name);
};
bool Integer::operator [](const char *name)
{
      if ((strcmp(name, "first") == 0) && ((value & 1) != 0))
            return true;
      if ((strcmp(name, "second") == 0) && ((value & 2) != 0))
            return true;
      return false;
}
void main()
{
      Integer n1(2);
      bool ret = n1["second"];
}
```

Operators for classes

- One can also overload the index operator [] (to be used with different keys). The following example uses ***operator[]*** with both ***int*** and ***const char **** keys.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator [](const char *name);
	bool operator [](int index);
};
bool Integer::operator [](int index)
{
	...
}
bool Integer::operator [](const char *name)
{
	...
}
void main()
{
	Integer n1(2);
	bool ret = n1["second"];
	bool v2 = n1[2];
}
```

Operators for classes

- The function call operator (***operator()***)works almost the same as the indexing operator.
- Like the indexing operator, the function call operator () can only be a member function within the class

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	bool operator ()(int index);
};
bool Integer::operator ()(int index)
{
	return (value & (1 << index)) != 0;
}
void main()
{
	Integer n1(2);
	bool ret = n1(1);
}
```

Operators for classes

- The main difference between index operator (***operator[]***) and function call operator (***operator()***) is that function call operator can have multiple parameters (or none).

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator ()(int start,int end);
};
int Integer::operator ()(int start,int end)
{
	return (value >> start) & ((1 << (end - start)) - 1);
}

void main()
{
	Integer n1(122);
	int res = n1(1,3);
}
```

Operators for classes

- In this example, the function call operator (***operator()***) is used without any parameter:

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator ()();
};
int Integer::operator ()()
{
	return (value*2);
}

void main()
{
	Integer n1(122);
	int res = n1();
}
```

Operators for classes

- The member access operator (***operator→***) can also be overwritten.
- In this case, even though there are no restrictions imposed by the compiler, this operator has to return a pointer to an object.

App.cpp

```
class MyData
{
	float value;
public:
	void SetValue(float val) { value = val; }
};
class Integer
{
	MyData data;
public:
	MyData* operator -> ();
};
MyData* Integer::operator ->()
{
	return &data;
}
void main()
{
	Integer n1;
	n1->SetValue(100);
}
```

Operators for classes

- *operator→* has to be used with an object (NOT a pointer). The following example will not compile as ***n2*** (a pointer) does not have a data member called ***SetValue***.

App.cpp

```
class MyData
{
	float value;
public:
	void SetValue(float val) { value = val; }
};
class Integer
{
	MyData data;
public:
	MyData* operator -> ();
};
MyData* Integer::operator ->()
{
	return &data;
}
void main()
{
	Integer n1;
	Integer *n2 = &n1;
	n2->SetValue(100);
}
```

Operators for classes

- However, if we convert the pointer to an object, the **operator→** will work.
- The "→" operator can be defined only in a class (it cannot be defined outside the class as a friend function)

App.cpp

```
class MyData
{
      float value;
public:
      void SetValue(float val) { value = val; }
};
class Integer
{
      MyData data;
public:
      MyData* operator -> ();
};
MyData* Integer::operator ->()
{
      return &data;
}
void main()
{
      Integer n1;
      Integer *n2 = &n1;
      (*n2)->SetValue(100);
}
```

Operators for classes

- Other operators that behave in the same way as ***operator→*** are:
  - . (A.B)
  - **->* (A->*B)**
  - **.* (A.*B)**
  - *** (*A)**
  - **& (&A)**

Operators for classes

- The list operator ***“operator,”*** is used in case of lists
- For example, the following list is evaluated from left to right and without a specific operator, it returns the last value::

```
int x = (10,20,30,40)
```

- The evaluation is done as follows:
  - First evaluated is the expression “10,20” → which returns 20
  - The following expression which is evaluated is “20,30” (20 returned from the previous expression) which returns 30
  - And finally, it is evaluated “30,40” which will return 40

Operators for classes

- In this case, the "***operator,***" is called first, for n1 and 2.5f, which returns 75 (30*2.5 = 75)
- ***res*** local variable will have the value 75

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator , (float f);
};
int Integer::operator ,(float f)
{
	return (int)(value*f);
}

void main()
{
	Integer n1(30);
	int res = (n1, 2.5f);
}
```

Operators for classes

- In this case the "***operator,***" is called first for ***n1*** and ***2.5f***, which returns the value 30*2.5=75, then the default "," operator for 75 and 10 is applied that returns 10.

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	int operator , (float f);
};
int Integer::operator ,(float f)
{
	return (int)(value*f);
}

void main()
{
	Integer n1(30);
	int res = (n1, 2.5f, 10 );
}
```

Operators for classes

- It is recommended to use ***friend*** specifier to explain several combinations that can be found when using the list operator (***operator,***)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend int operator , (Integer&, float f);
	friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
	return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
	return value + i.value;
}
void main()
{
	Integer n1(30);
	int res = (n1, 2.5f, n1);
}
```

Operators for classes

- It is recommended to use ***friend*** specifier to explain several combinations that can be found when using the list operator (***operator,***)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend int operator , (Integer&, float f);
	friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
	return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
	return value + i.value;
}
void main()
{
	Integer n1(30);
	int res = (n1, 2.5f, n1);
}
```

(n1,2.5f)=75

Operators for classes

- It is recommended to use ***friend*** specifier to explain several combinations that can be found when using the list operator (***operator,***)

App.cpp

```
class Integer
{
	int value;
public:
	Integer(int val) : value(val) {}
	friend int operator , (Integer&, float f);
	friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
	return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
	return value + i.value;
}
void main()
{
	Integer n1(30);
	int res = (n1, 2.5f, n1);
}
```

(75,n1) = 105

Operators for classes

- When overloading an operator, some optimizations that compiler is doing (such as lazy evaluation) will be lost.

App.cpp

```
class Bool {
    bool value;
    const char * name;
public:
    Bool(bool val, const char * nm) : value(val), name(nm) {}
    Bool operator|| (const Bool &i) {
        bool res = value || i.value;
        printf("Compute bool(%s and %s)=>%s\n", name, i.name, res ? "true" : "false");
        Bool b(res, "temp");
        return b;
    };
    operator bool() {
        printf("Return bool for %s => %s\n", this->name,value?"true":"false");
        return value;
    }
};
int main() {
    Bool n1(true,"n1");
    Bool n2(false,"n2");
    Bool n3(false,"n3");
    bool res = ((bool)n1) || ((bool)n2) || ((bool)n3);
    return 0;
}
```

Output:

Return bool for n1 => true

Once **n1** is evaluated and it is **true**, the rest of the evaluation is skipped. Lazy evaluation for || operator is applied and the cast operators for **n2** and **n3** are not called anymore.

Operators for classes

- When overloading an operator, some optimizations that compiler is doing (such as lazy evaluation) will be lost.

App.cpp

```
class Bool {
    bool value;
    const char * name;
public:
    Bool(bool val, const char * nm) : value(val), name(nm) {}
    Bool operator|| (const Bool &i) {
        bool res = value || i.value;
        printf("Compute bool(%s || %s)=>%s\n", name, i.name, res ? "true" : "false");
        Bool b(res, "temp");
        return b;
    };
    operator bool() {
        printf("Return bool for %s => %s\n", this->name,value?"true":"false");
        return value;
    }
};
int main() {
    Bool n1(true,"n1");
    Bool n2(false,"n2");
    Bool n3(false,"n3");
    bool res = n1 || n2 || n3;
    return 0;
}
```

Output:

Compute bool(n1 || n2)=>true
Compute bool(temp || n3)=>true
Return bool for temp => true

However, in this case since we have used our own **operator||** the lazy evaluation will not be applied, and the entire expression will be evaluated.

- Operations with objects

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
};
void Set(Date d,int value)
{
	d.SetX(value);
}
void main()
{
	Date d(1);
	Set(d,123);
}
```

```
push    123
sub     esp,10h
mov     eax,esp
mov     ecx,dword ptr [d.X]
mov     dword ptr [eax],ecx
mov     edx,dword ptr [d.Y]
mov     dword ptr [eax+4],edx
mov     ecx,dword ptr [d.Z]
mov     dword ptr [eax+8],ecx
mov     edx,dword ptr [d.T]
mov     dword ptr [eax+0Ch],edx
call    Set
add     esp,14h
```

10h = 16 = sizeof(Data)

- In this case, as there is no copy-constructor the compiler copies the entire object "d" byte-by-byte similar to what ***memcpy*** function does.

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
    int X, Y, Z, T;
public:
    Date(int value) : X(value), Y(value + 1),
        Z(value + 2), T(value + 3) {}
    Date(const Date & d) : X(d.X), Y(d.Y),
        Z(d.Z), T(d.T) {}
    void SetX(int value) { X = value; }
};
void Set(Date d, int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d, 123);
}
```

```
push    123

sub     esp,10h

mov     ecx,esp
lea     eax,[d]
push    eax
call    Date::Date
add     esp,14h

call    Set
add     esp,14h
```

10h = 16 = sizeof(Data)

Copy Constructor

- However, if a copy-constructor exists, that method will be call to copy the object "d" into the stack.

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
    int X, Y, Z, T;
public:
    Date(int value) : X(value), Y(value + 1),
        Z(value + 2), T(value + 3) {}
    Date(Date && d) : X(d.X), Y(d.Y),
        Z(d.Z), T(d.T) {}
    void SetX(int value) { X = value; }
};
void Set(Date d, int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d, 123);
}
```

Move Constructor

error C2280: 'Date::Date(const Date &)': attempting to reference a deleted function
note: compiler has generated 'Date::Date' here
note: 'Date::Date(const Date &)': function was implicitly deleted because 'Date' has a user-defined move constructor

- If the move constructor is present, but not a copy constructor, the code will fail at compile time. Adding a copy-constructor will make this code work.

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
      int X,Y,Z,T;
public:
      Date(int value) : X(value), Y(value + 1),
                        Z(value + 2), T(value+3) {}
      void SetX(int value) { X = value; }
};
void Set(Date &d,int value)
{
      d.SetX(value);
}
void main()
{
      Date d(1);
      Set(d,123);
}
```

```
push        123
lea         eax,[d]
push        eax
call        Set
add         esp,8
```

- If we send the object via a reference or pointer, the copy-constructor is no longer required.

Object operations

When a parameter is given to a function, we have the following cases:

- If the parameter is ***reference/pointer*** – only its address is copied on the stack

- If the parameter is an ***object***, all of that object is copied on the stack and it can be accessed as any other parameter that was copied on the stack (relative at [EBP+xxx]). The compiler uses the copy-constructor to copy an object into the stack. If no copy constructor is present, a ***memcpy*** – like code is generated (a code that copies byte-by-byte the entire content of the object into the stack).

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
};
Date Get(int value)
{
	Date d(value);
	return d;
}
void main()
{
	Date d(1);
	d = Get(100);
}
```

```
Date Get(int value)
{
	push	ebp
	mov	ebp,esp
Date d(value);
	mov	eax,dword ptr [value]
	push	eax
	lea	ecx,[d]
	call	Date::Date
return d;
	mov	eax,dword ptr [ebp+8]
	mov	ecx,dword ptr [d.X]
	mov	dword ptr [eax],ecx
	mov	edx,dword ptr [d.Y]
	mov	dword ptr [eax+4],edx
	mov	ecx,dword ptr [d.Z]
	mov	dword ptr [eax+8],ecx
	mov	edx,dword ptr [d.T]
	mov	dword ptr [eax+12],edx
	mov	eax,dword ptr [ebp+8]
}
	mov	esp,ebp
	pop	ebp
	ret
```

- In this case, function ***Get*** returns an object (NOT a reference) of type *Date*.

Object operations

- Let's analyze the following code :

App.cpp

```
class Date
{
      int X,Y,Z,T;
public:
      Date(int value) : X(value), Y(value + 1),
                        Z(value + 2), T(value+3) {}
      void SetX(int value) { X = value; }
};
Date Get(int value)
{
      Date d(value);
      return d;
}
void main()
{
      Date d(1);
      d = Get(100);
}
```

```
Date d(1);
      push        1
      lea         ecx,[d]
      call        Date::Date
d = Get(100);
      push        64h
      lea         eax,[ebp-0F0h]
      push        eax
      call        Get
      add         esp,8
      mov         ecx,dword ptr [eax]
      mov         dword ptr [d.X],ecx
      mov         edx,dword ptr [eax+4]
      mov         dword ptr [d.Y],edx
      mov         ecx,dword ptr [eax+8]
      mov         dword ptr [d.Z],ecx
      mov         edx,dword ptr [eax+0Ch]
      mov         dword ptr [d.T],edx
}
```

- In this case, function ***Get*** returns an object (NOT a reference) of type *Date*.

Object operations

- Let’s analyze the following code :

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
};
Date Get(int value)
{
	Date d(value);
	return d;
}
void main()
{
	Date d(1);
	d = Get(100);
}
```

App.pseudocode

```
class Date
{
 ...
};

Date* Get(Date *tempObject, int value)
{
	Date d(value);
	memcpy(tempObject,&d,sizeof(Date));
	return tempObject;
}
void main()
{
	Date d(1);
	unsigned char temp[sizeof(Date)]
	Date* tmpObj = Get(temp,100);
	memcpy(d,tmpObj,sizeof(Date))
}
```

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
	Date(const Date & obj) { X = obj.X; }
};
Date Get(int value)
{
	Date d(value);
	return d;
}
void main()
{
	Date d(1);
	d = Get(100);
}
```

App.pseudocode

```
class Date
{
 ...
};

Date* Get(Date *tempObject, int value)
{
	Date d(value);
	tempObject->Date(d);
	return tempObject;
}
void main()
{
	Date d(1);
	unsigned char temp[sizeof(Date)]
	Date* tmpObj = Get(temp,100);
	memcpy(d,tmpObj,sizeof(Date))
}
```

Object operations

- Let’s analyze the following code:

App.cpp

```
class Date
{
      int X,Y,Z,T;
public:
      Date(int value) : X(value), Y(value + 1),
                        Z(value + 2), T(value+3) {}
      void SetX(int value) { X = value; }
      Date(const Date & obj) { X = obj.X; }
      Date& operator = (const Date &d)
      {
            X = d.X;
            return (*this);
      }
};
Date Get(int value)
{
      Date d(value);
      return d;
}
void main()
{
      Date d(1);
      d = Get(100);
}
```

App.pseudocode

```
class Date
{
       ...
};

Date* Get(Date *tempObject, int value)
{
      Date d(value);
      tempObject->Date(d);
      return tempObject;
}
void main()
{
      Date d(1);
      unsigned char temp[sizeof(Date)]
      Date* tmpObj = Get(temp,100);
      d.operator=(*tmpObj);
}
```

Object operations

- Let's analyze the following code:

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
	Date(const Date & obj) { X = obj.X; }
	Date(const Date && obj) { X = obj.X; }
	Date& operator = (const Date &d)
	{
		X = d.X;
		return (*this);
	}
};
Date Get(int value)
{
	Date d(value);
	return d;
}
void main()
{
	Date d(1);
	d = Get(100);
}
```

App.pseudocode

```
class Date
{
	...
};

Date* Get(Date *tempObject, int value)
{
	Date d(value);
	tempObject->Date(d);
	return tempObject;
}
void main()
{
	Date d(1);
	unsigned char temp[sizeof(Date)]
	Date* tmpObj = Get(temp,100);
	d.operator=(*tmpObj);
}
```

If present, the move constructor is used when returning an object from a function !

Object operations

When dealing with temporary object, the compiler will always prefer:

1. Move constructor
2. Move assignment

Instead of copy constructor or simple assignment.

These methods (move constructor / move assignment) are preferred if they exist. If they are not specified, the copy-constructor and assignment operator will be used (if any).

Move constructor and Move assignment are only used for temporary object. For a regular object (reference) copy-constructor and assignment operator are preferred (if they exists).

Object operations

- Let’s consider the following class:

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) {
        text = new char[strlen(string) + 1];
        memcpy(text, string, strlen(string) + 1);
    }
public:
    String(const char * s) {
        CopyString(s);
        printf("CTOR => Obj:%p,Text:%p\n", this,text);
    }
    ~String() {
        if (text != nullptr) {
            printf("DTOR => Obj:%p,Text:%p\n", this, text);
            delete text;
        } else {
            printf("DTOR => Obj:%p (nothing to delete)\n", this);
        }
    }
}
```

Object operations

- Let’s consider the following class:

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { … }
public:
    String(const char * s) { … }
    ~String() { … }

    String(const String & obj) {
        CopyString(obj.text);
        printf("COPY => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text,&obj,obj.text);
    }
    String& operator = (const String &obj) {
        if (text != nullptr) {
            printf("Clear => Obj:%p,Text:%p\n", this, text);
            delete text;
            text = nullptr;
        }
        CopyString(obj.text);
        printf("EQ(Copy) => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text, &obj, obj.text);
        return (*this);
    }
}
```

- We will also add a copy-constructor and an assignment operator to this class.

Object operations

- What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { … }
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) { … }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

Object operations

- What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { … }
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) { … }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

```
Entering main function
CTOR => Obj:010FFA68,Text:01295040
Entering Get function
CTOR => Obj:010FFA04,Text:01295070
Exiting Get function
COPY => Obj:010FFA24,Text:01294EF8 from Obj:010FFA04,Text:01295070
DTOR => Obj:010FFA04,Text:01295070
Clear => Obj:010FFA68,Text:01295040
EQ(Copy) => Obj:010FFA68,Text:01294F30 from Obj:010FFA24,Text:01294EF8
DTOR => Obj:010FFA24,Text:01294EF8
Exiting main function
DTOR => Obj:010FFA68,Text:01294F30
```

Translated

```
Entering main function
CTOR => main::s (Text:01295040)
    Entering Get function
    CTOR => Get::s (Text:01295070)
    Exiting Get function
    COPY => temp_obj_1(Text:01294EF8) from Get::s(Text:01295070)
    DTOR => Get::s(Text:01295070)
Clear => main::s(Text:01295040)
EQ(Copy) => main::s(Text:01294F30) from temp_obj_1(Text:01294EF8)
DTOR => temp_obj_1(Text:01294EF8)
Exiting main function
DTOR => main::s(Text:01294F30)
```

GET function

Object operations

- What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { …
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) …
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

From the ***String::text*** point of view the following happen:

1. Allocate memory for main::s::text
2. Allocate memory for Get::s::text
3. Allocate memory for temp_obj_1::text
4. Copy string from Get::s::text to temp_obj_1::text
5. Free memory for Get::s::text
6. Free memory for main::s::text
7. Copy memory from temp_obj_1::text to main::s::text
8. Free memory for temp_obj_1::text
9. Free memory for main::s::text

"C++ test" is allocated 3 times (for Get::S, temp_obj_1 and main::s)

Translated

GET function

```
Entering main function
CTOR => main::s (Text:01295040)
Entering Get function
CTOR => Get::s (Text:01295070)
Exiting Get function
COPY => temp_obj_1(Text:01294EF8) from Get::s(Text:01295070)
DTOR => Get::s(Text:01295070)
Clear => main::s(Text:01295040)
EQ(Copy) => main::s(Text:01294F30) from temp_obj_1(Text:01294EF8)
DTOR => temp_obj_1(Text:01294EF8)
Exiting main function
DTOR => main::s(Text:01294F30)
```

Object operations

- Let's consider the following class:

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { … }
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) { … }

    String(String && obj) {
        this->text = obj.text;
        obj.text = nullptr;
        printf("MOVE => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text, &obj, obj.text);
    }
    String& operator = (String &&obj) {
        this->text = obj.text;
        printf("EQ(Move) => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text, &obj, obj.text);
        obj.text = nullptr;
        return (*this);
    }
}
```

- Now, we will add a move constructor and a move assignment to the class as well.

Object operations

- What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { … }
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) { … }
    String(String && obj) { … }
    String& operator = (String &&obj) { … }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

Object operations

- What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { … 
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) { .
    String(String && obj) { … }
    String& operator = (String &&obj) { … }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

```
Entering main function
CTOR => Obj:00AFFBD0,Text:00DA60A0
Entering Get function
CTOR => Obj:00AFFB6C,Text:00DA60D0
Exiting Get function
MOVE => Obj:00AFFB8C,Text:00DA60D0 from Obj:00AFFB6C,Text:00000000
DTOR => Obj:00AFFB6C (nothing to delete)
EQ(Move) => Obj:00AFFBD0,Text:00DA60D0 from Obj:00AFFB8C,Text:00DA60D0
DTOR => Obj:00AFFB8C (nothing to delete)
Exiting main function
DTOR => Obj:00AFFBD0,Text:00DA60D0
```

Translated

```
Entering main function
CTOR => main::s(Text:00DA60A0)
  Entering Get function
  CTOR => Get::s(Text:00DA60D0)
  Exiting Get function
  MOVE => temp_obj_1(Text:00DA60D0) from Get::s(Text:00000000)
  DTOR => Get::s (nothing to delete)
EQ(Move) => main::s,Text:00DA60D0 from temp_obj_1(Text:00DA60D0)
DTOR => temp_obj_1 (nothing to delete)
Exiting main function
DTOR => main::s(Text:00DA60D0)
```

GET function

Object operations

- What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { …
public:
    String(const char * s) { … }
    ~String() { … }
    String(const String & obj) { … }
    String& operator = (const String &obj) …
    String(String && obj) { … }
    String& operator = (String &&obj) { … }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

<u>From the ***String::text*** point of view the following happen:</u>

1. Allocate memory for main::s::text
2. Allocate memory for Get::s::text
3. Copy the pointer of Get::s::text to temp_obj_1::text
4. Make Get::s::text NULL (nullptr)
5. Copy the pointer of temp_obj_1::text to main::s::text
6. Make temp_obj_1::text NULL (nullptr)
7. Free memory for main::s::text

"C++ test" is allocated one time and then the pointer is moved until in reaches the destination object ("s" from main)

Translated

```
Entering main function
CTOR => main::s(Text:00DA60A0)
    Entering Get function
    CTOR => Get::s(Text:00DA60D0)
    Exiting Get function
    MOVE => temp_obj_1(Text:00DA60D0) from Get::s(Text:00000000)
    DTOR => Get::s (nothing to delete)
EQ(Move) => main::s,Text:00DA60D0 from temp_obj_1(Text:00DA60D0)
DTOR => temp_obj_1 (nothing to delete)
Exiting main function
DTOR => main::s(Text:00DA60D0)
```

GET function

Object operations

- What is the problem with the following code ?

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
	Date(const Date & obj) { X = obj.X; }
	Date& operator = (Date &d)
	{
		X = d.X;
		return (*this);
	}
};
Date& Get(int value)
{
	Date d(value);
	return d;
}
void main()
{
	Date d(1);
	d = Get(100);
}
```

App.pseudocode

```
class Date
{
	...
};
Date* Get(int value)
{
	Date d(value);
	return &d;
}
void main()
{
	Date d(1);
	Date* tmpObj = Get(100);
	d.operator=(*tmpObj);
}
```

Object operations

- What is the problem with the following code ?

App.cpp

```
class Date
{
	int X,Y,Z,T;
public:
	Date(int value) : X(value), Y(value + 1),
	                  Z(value + 2), T(value+3) {}
	void SetX(int value) { X = value; }
	Date(const Date & obj) { X = obj.X; }
	void operator = (Date &d)
	{
		X = d.X;
	}
};
Date& Get(int value)
{
	Date d(value);
	return d;
}
void main()
{
	Date d(1);
	d = Get(100);
}
```

```
void operator = (Date &d)
{
	push        ebp
	mov         ebp,esp
	sub         esp,0CCh
	push        ebx
	push        esi
	push        edi
	push        ecx
	lea         edi,[ebp-0CCh]
	mov         ecx,33h
	mov         eax,0CCCCCCCCh
	rep stos    dword ptr es:[edi]
	pop         ecx
	mov         dword ptr [this],ecx
X = d.X;
	mov         eax,dword ptr [this]
	mov         ecx,dword ptr [d]
	mov         edx,dword ptr [ecx]
	mov         dword ptr [eax],edx
}
	pop         edi
	pop         esi
	pop         ebx
	mov         esp,ebp
	pop         ebp
	ret         4
```

Q & A