CLASSES (II)

Overloading Operators

```
1 int a, b, c;
2 a = b + c;
```

Here, different variables of a fundamental type (int) are applied the addition operator, and then the assignment operator.
For a fundamental arithmetic type, the meaning of such operations is generally obvious and unambiguous, but it may not be so for certain **class** types.

```
1 struct myclass {
2   string product;
3   float price;
4 } a, b, c;
5 a = b + c;
```

Here, it is not obvious what the result of the addition operation on b and c does. In fact, this code alone would cause a compilation error, since the type **myclass** has no defined behavior for additions.

However, C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes.
Here is a list of all the operators that can be overloaded:

Overloadable operators												
+	-	*	/	=	<	>	+=	-=	*=	/=	<<	>>
<<=	>>=	==	!=	<=	>=	++	--	%	&	^	!	|
~	&=	^=	|=	&&	||	%=	[]	()	,	->*	->	new
delete		new[]		delete[]								

Operators are overloaded by means of **operator** functions, which are regular functions with special names: their name begins by the ***operator*** *keyword* followed by the ***operator sign*** that is overloaded.

Cartesian vectors are sets of two coordinates: x and y.
The addition operation of two ***cartesian vectors*** is defined as the addition both x coordinates together, and both y coordinates together.
For example, adding the ***cartesian vectors*** (3,1) and (1,2) together would result in (3+1,1+2) = (4,3).

```cpp
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

```
4,3
```

```cpp
CVector (int, int) : x(a), y(b) {}  // function name CVector (constructor)
CVector operator+ (const CVector&); // function that returns a CVector
```

The function **operator+** of class Cvector overloads the addition **operator (+)** for that type. Once declared, this function can be called either implicitly using the operator, or explicitly using its functional name. Both expressions are equivalent.

```cpp
c = a + b;
c = a.operator+ (b);
```

The operator overloads are just regular functions which **can have any behavior**; there is actually no requirement that the operation performed by that overload bears a relation to the mathematical or usual meaning of the operator, although it is strongly recommended.
For example, a class that overloads **operator+** to actually subtract or that overloads **operator==** to fill the object with zeros, is perfectly valid, although using such a class could be challenging.

The parameter expected for a member function overload for operations such as **operator+** is naturally the operand to the right hand side of the operator.

This is common to all binary operators (those with an operand to its left and one operand to its right).

But operators can come in diverse forms. Here you have a table with a summary of the parameters needed for each of the different operators than can be overloaded (please, replace @ by the operator in each case):

Expression	Operator	Member function	Non-member function
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A,int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A,B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= []	A::operator@(B)	-
a(b,c...)	()	A::operator()(B,C...)	-
a->b	->	A::operator->()	-
(TYPE) a	TYPE	A::operator TYPE()	-

Where a is an object of class A, b is an object of class B and c is an object of class C. TYPE is just any type (that operators overloads the conversion to type TYPE).

Notice that some operators may be overloaded in two forms: either as a member function or as a non-member function: The first case has been used in the example above for **operator+**. But some operators can also be overloaded as non-member functions; In this case, the operator function takes an object of the proper class as first argument.

```
// non-member operator overloads
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {}
    CVector (int a, int b) : x(a), y(b) {}
};


CVector operator+ (const CVector& lhs, const CVector& rhs) {
  CVector temp;
  temp.x = lhs.x + rhs.x;
  temp.y = lhs.y + rhs.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

```
4,3
```

The Keyword THIS

The keyword **this** represents a pointer to the object **whose member function is being executed.** *It is used within a class's member function* **to refer to the object itself.**

One of its uses can be **to check if a parameter passed to a member function is the object itself.**

```cpp
// example on this
#include <iostream>
using namespace std;

class Dummy {
  public:
    bool isitme (Dummy& param);
};

bool Dummy::isitme (Dummy& param)
{
  if (&param == this) return true;
  else return false;
}

int main () {
  Dummy a;
  Dummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b\n";
  return 0;
}
```

```
yes, &a is b
```

It is also frequently used in `operator=` member functions that return objects by reference. Following with the examples on *cartesian vector* seen before, its `operator=` function could have been defined as:

```cpp
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

In fact, this function is very similar to the code that the compiler generates implicitly for this class for **operator=**.

Static Members

A class can contain static members, either data or functions.
A static data member of a class is also known as a "**class variable**", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., **its value is not different from one object of this class to another**.

It may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```cpp
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
  public:
    static int n;
    Dummy () { n++; };
};

int Dummy::n=0;

int main () {
  Dummy a;
  Dummy b[5];
  cout << a.n << '\n';
  Dummy * c = new Dummy;
  cout << Dummy::n << '\n';
  delete c;
  return 0;
}
```

In fact, static members **have the same properties as non-member variables** but they enjoy class scope.
For that reason, and ***to avoid them to be declared several times, they cannot be initialized directly in the class***, but need to be initialized somewhere outside it. As in the previous example:

```cpp
int Dummy::n=0;
```

Because ***it is a common variable value for all the objects of the same class***, it can be referred to as a member of any object of that class or even directly by the class name (of course this is only valid for static members):

```cpp
cout << a.n;
cout << Dummy::n;
```

These two calls above are referring to the same variable: *the static variable n* within class Dummy shared by all objects of this class.
Again, it is just like a non-member variable, but with a name that requires to be accessed like a member of a class (or an object).

Classes can also have static member functions. These represent the same: members of a class that are common to all object of that class, **acting exactly as non-member functions but being accessed like members of the class.** Because

they are like non-member functions, they cannot access non-static members of the class (neither member variables nor member functions). They neither can use the keyword **this**.

Const Member Functions

When an object of a class is qualified as a const object:

```
const MyClass myobject;
```

The access to its data members from outside the class is restricted to **read-only**, ***as if all its data members were const*** for those accessing them from outside the class.
Note though, that ***the constructor is still called and is allowed to initialize and modify these data members***:

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {
  const MyClass foo(10);
// foo.x = 20;            // not valid: x cannot be modified
  cout << foo.x << '\n';  // ok: data member x can be read
  return 0;
}
```

```
10
```

The member functions of a const object can only be called if they are themselves specified as const members; in the example above, member ***get (which is not specified as const)*** cannot be called from foo. To specify that a member is a **const** member, the **const keyword shall follow the function prototype**, after the closing parenthesis for its parameters:

```
int get() const {return x;}
```

Note that **const** can be used to qualify the type returned by a member function. *This **const** is not the same as the one which specifies a member as const*.
Both are independent and are located at different places in the function prototype:

```
int get() const {return x;}         // const member function
const int& get() {return x;}        // member function returning a const&
const int& get() const {return x;} // const member function returning a const&
```

Member functions specified to be **const** cannot modify non-static data members nor call other non-const member functions. In essence, **const** members shall not modify the state of an object.

const objects are *limited to access only member functions marked as* ***const***, but *non-const objects are not restricted and thus can access both const and non-const member functions* alike.

You may think that anyway you are seldom going to declare **const** objects, and thus marking all members that don't modify the object as **const** is not worth the effort, but **const objects** are actually very common.
Most functions taking classes as parameters actually **take them by const reference,** and thus, these functions can only access their const members:

```
// const objects
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
};

void print (const MyClass& arg) {
  cout << arg.get() << '\n';
}

int main() {
  MyClass foo (10);
  print(foo);

  return 0;
}
```

```
10
```

get was not specified as a const member, the call to **arg.get()** in the print function would not be possible, because **const** objects only have access to const member functions.

Member functions can be overloaded on their **constness**:
A class may have two member functions with identical signatures except that one is const and the other is not: in this case, the const version is called only when the object is itself const, and the non-const version is called when the object is itself non-const.

```cpp
// overloading members on constness
#include <iostream>
using namespace std;

class MyClass {
    int x;
  public:
    MyClass(int val) : x(val) {}
    const int& get() const {return x;}
    int& get() {return x;}
};

int main() {
  MyClass foo (10);
  const MyClass bar (20);
  foo.get() = 15;          // ok: get() returns int&
// bar.get() = 25;         // not valid: get() returns const int&
  cout << foo.get() << '\n';
  cout << bar.get() << '\n';

  return 0;
}
```

```
15
20
```