The second

Saturday, March 13, 2021 7:59 PM

Bùi Lê Phi Long	16619

Chapter 19 : Functions

What is it?

- A function is a block of code which only runs when it is called
- We can pass data, known as parameters into a function

Why we use it?

- Function are used to perfom certain actions, and they are important for reusing code: Dèine code once, and use it many times.

Create a Function

- C++ provides some pre-defined functions, such as main(), which is used to execute code, but we can also create our functions to perform certain actions

Syntax:

`type function_name(arguments) { statement; statement; return something; }`	type can be: void, int, float, double,… inside the ( ) *arguments* could be the parameters
`void myFunction() { // code to be executed }`	myFunction() is the name of the function void means that the function does not have a return value this function accepts no parameters inside the function, add the code that defines what the function should do.
`int mysquarednumber (int x); int main() { }`	declaring the function of type int accepting one parameter integer x.
`int mysum(int x, int y); int main() { }`	declaring a function of type int, accepts two int parameters
`int mysum(int, int); int main() { }`	! **<u>JUST IN FUNCTION DECLARATION ONLY,</u>** we can omit the parameter names, but we need to specify their types

Function definition

- To be called in a program, a function must be defined first.
- A function definition has everything a function declaration has and the body of a function

Syntax:

```
#include <iostream>

void myfunction(); // function declaration

int main()
{

}

// function definition
void myfunction() {
    std::cout << "Hello World from a function.";
}
```

myfunction is declared at first
then the definition of it is written below the int main function, which is absolutely legal.

```
int mysquarednumber(int x); // function declaration

int main()
{

}

// function definition
int mysquarednumber(int x) {
    return x * x;
}
```

defining a function accepts 1 parameter

Example:

```
#include <iostream>
void myfunction(); // function declaration
int main()
{
myfunction(); // a call to a function
}
// function definition
void myfunction() {
std::cout << "Hello World from a function.";
}
```

<u>Output</u>:

```
Hello World from a function.
```

<u>Comments</u>:

- the function *myfunction()* is first declared, then is called in the main function, and lastly defined as the action of print "Hello World from a function".
- The program works perfectly

```
#include <iostream>
int mysquarednumber(int x); // function declaration
int main()
{
int myresult = mysquarednumber(2); // a call to the functio
std::cout << "Number 2 squared is: " << myresult;
}
// function definition
int mysquarednumber(int x) {
return x * x;
}
```

<u>Output:</u>

```
Number 2 squared is: 4
```

<u>Comments:</u>

- this time we create a function that accepts 1 parameter integer x. It works the same as the above program, *mysquarednumber()* accepts the integer x as parameter and return the value of x*x means x^2.

```
#include <iostream>
int mysum(int x, int y);
int main()
{
int myresult = mysum(5, 10);
std::cout << "The sum of 5 and 10 is: " << myresult;
}
int mysum(int x, int y) {
return x + y;
}
```

<u>Output</u>

```
The sum of 5 and 10 is: 15
```

<u>Comments:</u>

mysum() accepts 2 parameters those are integer x and y, and returns the value of the sum (x+y). inside *int main()* function, the variable *mysum* receive the value of the sum(5+10) through the function *mysum(5,10).*

Return statements

What is this? and why we need to return statements?

- Functions are of a certain type, also referred to as return type, and they must return a value
- The value returned is specified by a return-statement
- Functions of type void do not need a return statement

Example

```
#include <iostream>
void voidfn();
int main()
{
voidfn();
}
void voidfn()
{
std::cout << "This is void function and needs no return.";
}
int intfn()
{
return 42; // return statement
}
```

Output:

```
This is void function and needs no return.
```

Comments:

the *type void* function doesn't need to return any value, it just only perfom some action inside, in this example, the function *voidfn()* has print out some strings, and didn't return anything.

```
#include <iostream>
int multiplereturns(int x);
int main()
{
std::cout << "The value of a function is: " <<
multiplereturns(25);
}
int multiplereturns(int x)
{
if (x >= 42)
{
return x;
}
return 0;
```

Output:

```
The value of a function is: 0
```

Comments:

A function can have multiple return-statements if required. Once any of the returnstatement is executed, the function stops, and the rest of the code in the function is ignored.
In this example, the *function multiplereturns* accepts parameter x =25 then perform the action of passing it through the if-else function. Because x = 25 means x <42, then this function returned value of 0 and ignored the line return x above.

Passing Arguments

1. Passing by Value/Copy

When we pass an argument to a function, a copy of that argument is made and passed to the function if the function parameter type is not a reference. This means the value of the original argument does not change. A copy of the argument is made. Example:

```
#include <iostream>
void myfunction(int byvalue)
{
std::cout << "Argument passed by value: " << byvalue;
}
int main()
{
myfunction(123);
}
```

Output:

```
Argument passed by value: 123
```

Comments:

- The value of integer 123 was passed through the function, then the function created a copy of the value 123 then stored it in the *byvalue* variable, and then passed it throught the cout command.
- This is known as passing an argument *by value* or passing an argument *by copy*

2. Passing by Reference

When a function parameter type is a reference type, then the actual argument is passed to the function. The function can modify the value of the argument. Example:

```
#include <iostream>
void myfunction(int& byreference)
{
byreference++; // we can modify the value of the argument
std::cout << "Argument passed by reference: " << byreference;
}
int main()
{
int x = 123;
myfunction(x);
}
```

Output:

```
Argument passed by reference: 124
```

Comments:

- Here we passed an argument of a reference type int&, so the function now works with the actual argument and can change its value. When passing by reference, we need to pass the variable itself; we can't pass in a literal representing a value. Passing byreference is best avoided.
- reference of variable x passed through the *myfunction,* then x is modifed and printed out

3. Passing by Const Reference

Why would we have to pass the const reference ?

- What is preferred is passing an argument by *const reference,* also referred to as a *reference to const*. It can be more efficient to pass an argument by reference, but to ensure it is not changed, we make it of const reference type. Example

```
#include <iostream>
#include <string>
void myfunction(const std::string& byconstreference)
{
std::cout << "Arguments passed by const reference: " <<
byconstreference;
}
int main()
{
std::string s = "Hello World!";
myfunction(s);
}
```

Output:

```
Arguments passed by const reference: Hello World!
```

Comments:

We use passing by const reference for efficiency reasons, and the const modifier
ensures the value of an argument will not be changed.

```
#include <iostream>
#include <string>
void myfunction(const std::string& byconstreference);
int main()
{
std::string s = "Hello World!";
myfunction(s);
}
void myfunction(const std::string& byconstreference)
{
std::cout << "Arguments passed by const reference: " <<
byconstreference;
}
```

Output:

```
Arguments passed by const reference: Hello World!
```

Comments:

- Although a function definition is also a declaration, we should provide both the declaration and a definition.

Function overloading

What is this ???

- We can have multiple functions with the same name but with different parameter types.
- This called function overloading

```
void myprint(char param);
void myprint(int param);
void myprint(double param);
```

Why?

- When the funtion names are the same, but the parameter types differ, then we have overloaded functions

Example:

```
void myprint(char param);
void myprint(int param);
void myprint(double param);
#include <iostream>
void myprint(char param);
void myprint(int param);
void myprint(double param);
int main()
{
myprint('c'); // calling char overload
myprint(123); // calling integer overload
myprint(456.789); // calling double overload
}
void myprint(char param)
{
std::cout << "Printing a character: " << param << '\n';
}
void myprint(int param)
{
std::cout << "Printing an integer: " << param << '\n';
}
void myprint(double param)
{
std::cout << "Printing a double: " << param << '\n';
}
```

Output:

```
Printing a character: c
Printing an integer: 123
Printing a double: 456.789
```

Comments:

- When calling our functions, a proper overload is selected based on the type of argument we supply. In the first call to myprint('c'), a char overload is selected because literal 'c' is of type char. In a second function call myprint(123), an integer overload is selected because the type of an argument 123 is int. And lastly, in our last function call myprint(456.789), a double overload is selected by a compiler as the argument 456.789 is of type double.

Chapter 21 : Scope and Lifetime

What is it though?

When we declare a variable, its name is valid only inside some sections of the source code. And that section (part, portion, region) of the source code is called *scope*. It is the region of code in which the name can be accessed.

There are different scopes:

1. Local Scope

When we declare a name inside a function, that name has a *local scope*. Its scope starts from the point of declaration till the end of the function block marked with }.

```
void myfunction()
{
int x = 123; // Here begins the x's scope
} // and here it ends
```

Our variable ***x*** is declared inside a *myfunction()* body, and it has a local scope. We say that name x is local to *myfunction()*. It exists (can be accessed) only inside the function's scope and nowhere else.

2. Block Scope

The block-scope is a section of code marked by a block of code starting with { and ending with }. Example:

```
int main()
{
int x = 123; // first x' scope begins here
{
int x = 456; // redefinition of x, second x' scope begins here
} // block ends, second x' scope ends here
// the first x resumes here
} // block ends, scope of first x's ends here
```

3. Life time

The lifetime of an object is the time an object spends in memory. The lifetime is determined by a so-called *storage duration*. There are different kinds of storage durations.

4. Automatic Storage Duration

- The automatic storage duration is a duration where memory for an object is automatically allocated at the beginning of a block and deallocated when the code block ends.
- This also called a stack memory; objects are allocated on the stack. In this case, the object's life time is determined by its scope
- All local objects have this storage duration

5. Dynamic Storage Duration

- This duration is where memory for an object is manually allocated an manually deallocated.
- This kind of storage is often referred to as heap memory.
- The user determines when the memeory for an object will be allocated, and when it will be released.
- The life time of an object is not determined by a scope in which the object was defined.
- We do it through operator *new* and *smart pointers*, we should prefer smarter pointer facillities to operator new.

6. Static Storage Duration

- When an object declaration is prepended with a static specifier, it means the storage for a static object is allocated when the program starts and deallocated when the program ends.
- There is only one instance of such objects, and (with a few exceptions) their lifetime ends when a program ends.
- They are objects we can access at any given time during the execution of a program.

7. Operators new and delete

We can dynamically allocate and deallocate storage for our object and have pointers point to this newly allocated memory

The operator new allocates space for an object. The object is allocated on the *freestore*, often called *heap* or *heap memory*. The allocated memory must be deallocate using operator delete. It deallocates the memory previously allocated memory with an operator new. Example:

```
#include <iostream>
int main()
{
int* p = new int;
*p = 123;
std::cout << "The pointed-to value is: " << *p;
delete p;
}
```

Output:

```
The pointed-to value is: 123
```

Comments:
This example allocates space for one integer on the free-store. Pointer p now points to the newly allocated memory for our integer. We can now assign a value to our newly allocated integer object by dereferencing a pointer. Finally, we free the memory by calling the operator delete.

If we want to allocate memory for an array, we use the operator new[]. To deallocate a memory allocated for an array, we use the operator delete[]
Pointers and arrays
are similar and can often be used interchangeably. Pointers can be dereferenced by a subscript operator []
Example :

```
#include <iostream>
int main()
{
int* p = new int[3];
p[0] = 1;
p[1] = 2;
p[2] = 3;
std::cout << "The values are: " << p[0] << ' ' << p[1] << ' ' << p[2];
delete[] p;
}
```

Output:

```
The values are: 1 2 3
```

Comments:
This example allocates space for three integers, an array of three integers using operator new[]. Our pointer p now points at the first element in the array. Then, using asubscript operator [], we dereference and assign a value to each array element . Finally we deallocate the memory using operator delete[]

★ Remember:

always delete whatyou new-ed and always delete[] what you new[]-ed.

prefer *smart pointers* to operator new. The lifetime of objects allocated on the free-store is not bound by a scope in which the objects were defined. We manually allocate and manually deallocate the memory for our object, thus controlling when the object gets created and when it gets destroyed.

Chapter 23: Classes -Introduction

What is it?
Class is a user-defined type. A class consists of members. The members are data members and member functions. A class can be described as data and some functionality on that data, wrapped into one.

Syntax

class MyClass;	To only declare a class name
class MyClass{};	To define an empty class, we add a class body marked by braces {}
class MyClass{}; int main()	To create an instance of the class, an object, we use:

{
MyClass o;
}	

Why?
We defined a class called MyClass. Then we created an object o of type MyClass. It is said that o is an *object*, a *class instance*.

1. Data Member Fields

- A class can have a set of some data in it. These are called *member fields*.

Syntax

`class MyClass {char c; };`	One member field of the *class "MyClass"* is type of char
`class MyClass { char c; int x; double d; };`	Two more fields of type int and double: x and d Now our class has three member fields, and each member field has its name

2. Member Functions

What is it and why we use member functions ?
Similarly, a class can store functions. These are called *member functions*. They are mostly used to perform some operations on data fields
Syntax

```
class MyClass
{
void dosomething();
};
```

declaring a member function of type void called dosomething()

Declaring member function:

```
class MyClass
{
void dosomething()
{
std::cout << "Hello World from a class.";
}
};
```

We can define the function inside class

```
class MyClass
{
void dosomething();
};
void MyClass::dosomething()
{
std::cout << "Hello World from a class.";
}
```

We can also define the function out side the class. IN this case, we write the function type first-> followed by a class name -> followed by a scope resolution:: ioperator followed by a function name, list of parameters if any and a function body

```
class MyClass
{
void dosomething()
{
std::cout << "Hello World from a class.";
}
void dosomethingelse()
{
std::cout << "Hello Universe from a class.";
}
};
```

We can have multiple members functions in a class

```
class MyClass
{
void dosomething();
void dosomethingelse();
};
void MyClass::dosomething()
{
std::cout << "Hello World from a class.";
}
void MyClass::dosomethingelse()
{
std::cout << "Hello Universe from a class.";
}
```

And can have multiple declarations of functions inside of class, then
define it outside of class by using scope

```
class MyClass
{
int x;
void printx()
{
std::cout << "The value of x is:" << x;
}
};
```

This class has one data field of type int called x, and it has a member function called printx()

3. Access Specifiers

- **What is it ?**
  Acess specifiers specify access for class members
- **Why use it?**
  Wouldn't it be convenient if there was a way we could disable access to member fields but allow access to member functions for our object and other entities accessing our class members?
  There are three access specifiers/labels: public, protected, and private:

<pre>class MyClass
{
public:
// everything in here
// has public access level
protected:
// everything in here
// has protected access level
private:
// everything in here
// has private access level
};</pre>	

<pre>class MyClass
{
// everything in here
// has private access by default
};</pre>
	Default visibility/access specifier for a class is private if none of the access specifiers is present

<pre>struct MyStruct
{
// everything in here
// is public by default
};</pre>
	We can create class by struct, the difference is while class having every thing private by default, struct having it all in public access.

Example:

<pre>class MyClass
{
public:
int x;
void printx()
{
std::cout << "The value of x is:" << x;
}
};</pre>
	defining a class with public access specifiers

<pre>#include <iostream>
class MyClass
{
public:
int x;
void printx()
{
std::cout << "The value of data member x is: " << x;
}
};
int main()
{
MyClass o;
o.x = 123; // x is accessible to object o
o.printx(); // printx() is accessible to object o
}</pre>
	Output:
	
	2.66667
	
	Comments:
	Our object o now has direct access to all member fields as they are all marked public.
	Member fields always have access to each other regardless of the access specifier. That is
	why the member function printx() can access the member field x and print or change
	ts value.

<pre>#include <iostream>
class MyClass
{
private:
int x; // x now has private access
public:
void printx()
{
std::cout << "The value of x is:" << x; // x is accessible to
// printx()
}
};
int main()
{
MyClass o; // Create an object
o.x = 123; // Error, x has private access and is not accessible to
// object o
o.printx(); // printx() is accessible from object o
}</pre>
	Output:
	
	16 error: 'int MyClass::x' is private within this context
	
	Comments:
	Our object o now only has access to a member function printx() in the public
	section of the class. It cannot access members in the private section of the class.
	If we want the class members to be accessible to our object, then we will put them
	inside the public: area. If we want the class members not to be accessible to our object,
	then we will put them into the private: area.

4. Constructors

A constructor is a member function that has the same name as the class. To initialize an object of a class, we use constructors. Constructor's purpose is to initialize an object of a class. It constructs an object and can set values to data members. If a class has a constructor, all objects of that class will be initialized by a constructor call

1. Default constructor

```cpp
#include <iostream>
class MyClass
{
public:
MyClass()
{
std::cout << "Default constructor invoked." << '\n';
}
};
int main()
{
MyClass o; // invoke a default constructor
}
```

A constructor without parameters or with default parameters set is called a *defaultconstructor*. It is a constructor which can be called without arguments

```cpp
#include <iostream>
class MyClass
{
public:
MyClass(int x = 123, int y = 456)
{
std::cout << "Default constructor invoked." << '\n';
}
};
int main()
{
MyClass o; // invoke a default constructor
}
```

If a default constructor is not explicitly defined in the code, the compiler will generate a default constructor. But when we define a constructor of our own, the one that needs parameters, the default constructor gets removed and is not generated by a compiler.

Constructors are invoked when object initialization takes place. They can't be invoked directly.
Constructors can have arbitrary parameters; in which case we can call them *userprovided* constructors:

```cpp
#include <iostream>
class MyClass
{
public:
int x, y;
MyClass(int xx, int yy)
{
x = xx;
y = yy;
}
};
int main()
{
MyClass o{ 1, 2 }; // invoke a user-provided constructor
std::cout << "User-provided constructor invoked." << '\n';
std::cout << o.x << ' ' << o.y;
}
```

In this example, our class has two data fields of type int and a constructor. The constructor accepts two parameters and assigns them to data members. We invoke the constructor with by providing arguments in the initializer list with MyClass o{ 1, 2 };

★ Constructors do not have a return type, and their purposes are to initialize the object of its class.

2. Member Initialization

A better, more efficient way to initialize an object of a class is to use the constructor's *member initializer list* in the definition of the constructor:

```cpp
#include <iostream>
class MyClass
{
public:
int x, y;
MyClass(int xx, int yy)
: x{ xx }, y{ yy } // member initializer list
{ }
};
int main()
{
MyClass o{ 1, 2 }; // invoke a user-defined constructor
std::cout << o.x << ' ' << o.y;
}
```

Output:

```
1 2
```

Comments:

1,2 here is initialize as member of MyClass

A member initializer list starts with a colon, followed by member names and their initializers, where each initialization expression is separated by a comma. This is the preferred way of initializing class data members.

3. Copy Contructor

When we initialize an object with another object of the same class, we invoke a copy constructor. If we do not supply our copy constructor, the compiler generates a default copy constructor that performs the so-called shallow copy.

```
#include <iostream>
class MyClass
{
private:
int x, y;
public:
MyClass(int xx, int yy) : x{ xx }, y{ yy }
{ }
};
int main()
{
MyClass o1{ 1, 2 };
MyClass o2 = o1; // default copy constructor invoked
}
```

In this example, we initialize the object o2 with the object o1 of the same type. This
invokes the default copy constructor.

```
#include <iostream>
class MyClass
{
private:
int x, y;
public:
MyClass(int xx, int yy) : x{ xx }, y{ yy }
{ }
// user defined copy constructor
MyClass(const MyClass& rhs)
: x{ rhs.x }, y{ rhs.y } // initialize members with other object's
// members
{
std::cout << "User defined copy constructor invoked.";
}
};
int main()
{
MyClass o1{ 1, 2 };
MyClass o2 = o1; // user defined copy constructor invoked
}
```

Output:

```
User defined copy constructor invoked.
```

Comments:

Here we defined our own copy constructor in which we explicitly initialized data members with other objects data members, and we print out a simple message in the console / standard output

```
#include <iostream>
class MyClass
{
private:
int x;
int* p;
public:
MyClass(int xx, int pp)
: x{ xx }, p{ new int{pp} }
{ }
MyClass(const MyClass& rhs)
: x{ rhs.x }, p{ new int {*rhs.p} }
{
std::cout << "User defined copy constructor invoked.";
}
};
int main()
{
MyClass o1{ 1, 2 };
MyClass o2 = o1; // user defined copy constructor invoked
}
```

the default copy constructor does not *correctly* copy members of some types, such as pointers, arrays, etc. In order to properly make copies, we need to define our own copy logic inside the copy constructor

Here we have two constructors, one is a user-provided regular constructor, and the
other is a user-defined copy constructor. The first constructor initializes an object and is
invoked here: MyClass o1{ 1, 2 }; in our main function

4. Copy Assignment

We can also copy the values to an object after it has been initialized/created. We use a *copy assignment* for that.

```
MyClass copyfrom;
MyClass copyto = copyfrom; // on the same line, uses a copy constructor
```

When an object is created on one line and then assigned to in the next line, it then uses the *copy assignment* operator to copy the data from another object:

```
MyClass copyfrom;
MyClass copyto;
copyto = copyfrom; // uses a copy assignment operator
```

A copy assignment operator is of the following signature:

```
MyClass& operator=(const MyClass& rhs)
```

To define a user-defined copy assignment operator inside a class we use:

```
class MyClass
{
public:
MyClass& operator=(const MyClass& rhs)
{
// implement the copy logic here
return *this;
}
```

```
};
```

Notice that the overloaded = operators must return a dereferenced this pointer at the end. To define a user-defined copy assignment operator outside the class, we use:

```
class MyClass
{
public:
MyClass& operator=(const MyClass& rhs);
};
MyClass& MyClass::operator=(const MyClass& rhs)
{
// implement the copy logic here
return *this;
}
```

Similarly, there is a *move assignment* operator, which we will discuss later in the book. More on operator overloading in the following chapters.

5. Operator Overloading

Objects of classes can be used in expression as operands. For example, we can do the following:

myobject = otherobject;
myobject + otherobject;
myobject / otherobject;
myobject++;
++myobject;
	- These objects are used as operands.
	- To do this, we need to *overload* the operators for complex types such as classes
	- We need to overload them to provide a meaning ful operation on objects of a class
	- Some operators canbe overloaded,some cannot

- We can overload the following operators:

Arithmetic operators, binary operators, boolean operators, unary operators, comparison operators, compound operators, function and subscript operators:
+ - * / % ^ & | ~ ! = < > == != <= >= += -= *= /= %= ^= &= |= << >> >>= <<= && || ++ -- , ->* -> () []

Each operator carries its signature and set of rules when overloading for classes.
Some operator overloads are implemented as member functions, some as none member functions.

Example:

```
#include <iostream>
class MyClass
{
private:
int x;
double d;
public:
MyClass()
: x{ 0 }, d{ 0.0 }
{ }
// prefix operator ++
MyClass& operator++()
{
++x;
++d;
std::cout << "Prefix operator ++ invoked." << '\n';
return *this;
}
};
int main()
{
MyClass myobject;
// prefix operator
++myobject;
// the same as:
myobject.operator++();
}
```

Output:

```
Prefix operator ++ invoked.
Prefix operator ++ invoked.
```

Comments:
In this example, when invoked in our class, the overloaded prefix increment ++ operator increments each of the member fields by one.

That means, the line '*++myobject*'
has the same function as the '*myobject.operator++()*'

```cpp
#include <iostream>
class MyClass
{
private:
int x;
double d;
public:
MyClass()
: x{ 0 }, d{ 0.0 }
{ }
// prefix operator ++
MyClass& operator++()
{
    ++x;
++d;
std::cout << "Prefix operator ++ invoked." << '\n';
return *this;
}
// postfix operator ++
MyClass operator++(int)
{
MyClass tmp(*this); // create a copy
operator++(); // invoke the prefix operator overload
std::cout << "Postfix operator ++ invoked." << '\n';
return tmp; // return old value
}
};
int main()
{
MyClass myobject;
// postfix operator
myobject++;
// is the same as if we had:
myobject.operator++(0);
}
```

Output:

```
Prefix operator ++ invoked.
Postfix operator ++ invoked.
Prefix operator ++ invoked.
Postfix operator ++ invoked.
```

Comments:

Often operators depend on each other and can be implemented in terms of other operators. To implement a postfix operator **++**, we will implement it in terms of a prefix operator

```cpp
#include <iostream>
class MyClass
{
private:
int x;
double d;
public:
MyClass(int xx, double dd)
: x{ xx }, d{ dd }
{ }
MyClass& operator+=(const MyClass& rhs)
{
this->x += rhs.x;
this->d += rhs.d;
return *this;
}
};
int main()
{
MyClass myobject{ 1, 1.0 };
MyClass mysecondobject{ 2, 2.0 };
myobject += mysecondobject;
std::cout << "Used the overloaded += operator.";
}
```

Output:

```
Used the overloaded += operator.
Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

Comments:

myobject member field x has a value of 3, and a member field d has a value of 3.0.

```cpp
#include <iostream>
class MyClass
{
private:
int x;
double d;
public:
MyClass(int xx, double dd)
: x{ xx }, d{ dd }
{ }
MyClass& operator+=(const MyClass& rhs)
{
this->x += rhs.x;
this->d += rhs.d;
return *this;
}
friend MyClass operator+(MyClass lhs, const MyClass& rhs)
{
lhs += rhs;
return lhs;
}
```

Output:

```
Used the overloaded + operator.
```

Comments:
Like the program above but easier to code

```
};
int main()
{
MyClass myobject{ 1, 1.0 };
MyClass mysecondobject{ 2, 2.0 };
MyClass myresult = myobject + mysecondobject;
std::cout << "Used the overloaded + operator.";
}
```

6. Destructors

What is it ?

- Destructor is a member function that gets invoked when an object is destroyed

Syntax:

```
class MyClass
{
public:
MyClass() {} // constructor
~MyClass() {} // destructor
};
```

- Destructor takes no parameters, and thre is one destructer per class.

```
#include <iostream>
class MyClass
{
public:
MyClass() {} // constructor
~MyClass()
{
std::cout << "Destructor invoked.";
} // destructor
};
int main()
{
MyClass o;
} // destructor invoked here, when o gets out of scope
```

- Destructors are called when an object goes out of scope or when a pointer to an object is deleted. We should not call the destructor directly
- Destructors can be used to clean up the taken resources. Example:

`#include <iostream>`
`class MyClass`
`{`
`private:`
`int* p;`
`public:`
`MyClass()`
`: p{ new int{123} }`
`{`
`std::cout << "Created a pointer in the constructor." << '\n';`
`}`
`~MyClass()`
`{`
`delete p;`
`std::cout << "Deleted a pointer in the destructor." << '\n';`
`}`
`};`
`int main()`
`{`
`MyClass o; // constructor invoked here`
`} // destructor invoked here`
	Here we allocate memory for a pointer in the constructor and deallocate the memory in the destructor. This style of resource allocation/deallocation is called RAII or Resource
	Acquisition is Initialization. Destructors should not be called directly.

Chapter 24: Exercise

1. Class Instance

Write a program that defines an empty class called *MyClass* and makes an instance of *MyClass* in the main function.

`#include<iostream>`
`class MyClass`
`{`
`};`
`int main()`
`{`
`MyClass o;`
`}`
	Output: No output
	Comments:
	Initialize MyClass and an object o inside that class

2. Class with Data Members

Write a program that defines a class called *MyClass* with three data members of type char, int, and bool. Make an instance of that class inside the main function.

```cpp
#include<iostream>
class MyClass
{private:
     char c;
     int i;
     bool b;
};
int main()
{
MyClass o;
}
```

Output: No output
Comments:
Initializing class *MyClass* with the members of three type char integer and boolean

3. Class with Member Function

Write a program that defines a class called MyClass with one member function called printmessage(). Define the printmessage() member function inside the class and make it output the “Hello World” string. Create an instance of that class and use the object to call the class member function

```cpp
#include<iostream>
using namespace std;
class MyClass
{public:
     void printmessage()
     {
          cout<<"Hello World";
     }
};
int main()
{
MyClass o;
     o.printmessage();
}
```

Output:

```
Hello World
Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```

Comments:
Initializing class *MyClass* and declaring function in the public section, the printmessage() function will perfom the action of printing strings"Hello World".
- In the main function, creating an object 'o' then use o.printmessage() to perfom the action.

4. Class with Data and Function Members

Write a program that defines a class called MyClass with one member function called printmessage(). Define the printmessage() member function outside the class and have it output the "Hello World." string. Create an instance of that class and use the object to call the member function.

```cpp
#include<iostream>
using namespace std;
class MyClass
{public:
     void printmessage();
};
void MyClass :: printmessage()
{
     cout<<"Hello World";
}
int main()
{
MyClass o;
     o.printmessage();
}
```

```
Hello World
Process returned 0 (0x0)   execution time : 0.068 s
Press any key to continue.
```

5. Class Access Specifiers

Write a program that defines a class called MyClass with one private data member of type int called *x* and two member functions. The first member function called setx(int myvalue) will set the value of *x* to its parameter myvalue. The second member function is called getx(), is of type int and returns a value of *x*. Make an instance of the class and use the object to access both member functions.

```cpp
#include<iostream>
using namespace std;
class MyClass
{private:
     int x;
public:
     void setx(int myvalue)
     {
          x = myvalue;
     }
     void getx()
     {
          cout<<"The value of x : "<<x;
     }
};

int main()
{
MyClass o;
     o.setx(2);
     o.getx();
```

```
The value of x : 2
Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

6. User-defined Default Constructor and Destructor

Write a program that defines a class called MyClass with a user-defined default constructor and user-defined destructor. Define both constructor and destructor outside the class. Both member functions will output a free to choose the text on the standard output. Create an object of a class in function main.

```
#include<iostream>
using namespace std;
class MyClass
{
public:
    MyClass();
    ~MyClass();
};
MyClass::MyClass()
{
    cout<<"This is the constructor"<<endl;
};
MyClass::~MyClass()
{
    cout<<"This is the destructor"<<endl;
};
int main()
{
    MyClass o;

}
```

```
This is the constructor
This is the destructor

Process returned 0 (0x0)   execution time : 0.036 s
Press any key to continue.
```

7. Contructor Initializer List

Write a program that defines a class called MyClass, which has two private data members of type int and double. Outside the class, define a user-provided constructor accepting two parameters. The constructor initializes both data members with arguments using the initializer. Outside the class, define a function called printdata() which prints the values of both data members.

```
#include<iostream>
using namespace std;
class MyClass
{
private:
    int i;
    double d;
public:
    MyClass(int in, double dou);
    void printdata();
};
MyClass::MyClass(int in, double dou)
{
        i = in;
        d = dou;
}
void MyClass::printdata()
{
    cout<<"Integer is: "<<i<<" double is: "<<d;
}

int main()
{
    MyClass o(1,6.996);
    o.printdata();
}
```

```
Integer is: 1 double is: 6.996
Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

8. User-defined Copy Constructor

Write a program that defines a class called MyClass with arbitrary data fields. Write a user-defined constructor with parameters that initializes data members. Write a userdefined copy constructor which copies all the members. Make one object of the class called o1 and initialize it with values. Make another object of a class called o2 and initialize it with object o. Print data for both objects

```cpp
#include<iostream>
using namespace std;
class MyClass
{
private:
    int i;
    double d;
public:
    MyClass(int in, double dou);
    MyClass(const MyClass& rhs);
    void printdata();
};
MyClass::MyClass(int in, double dou)
{
        i = in;
        d = dou;
}
MyClass::MyClass(const MyClass& rhs)
{
    i = rhs.i;
    d = rhs.d;
}
void MyClass::printdata()
{
    cout<<"Integer is: "<<i<<" double is: "<<d;
}

int main()
{
    MyClass o(1,6.996);
    MyClass dak(6,6.9999999);
    dak = o;
    dak.printdata();
}
```

```
Integer is: 1 double is: 6.996
Process returned 0 (0x0)   execution time : 0.036 s
Press any key to continue.
```

9. User-defined Move Constructor

Write a program that defines a class with two data members, a user-provided constructor, a user-provided move constructor, and a member function that prints the data. Invoke the move constructor in the main program. Print the moved-to object data fields.

```cpp
#include<string>
using namespace std;
class MyClass
{
private:
    int i;
    double d;
public:
    MyClass(int in, double dou);
    MyClass(const MyClass&& move_object);
    void printdata();
};
MyClass::MyClass(int in, double dou)
{
        i = in;
        d = dou;
}
MyClass::MyClass(const MyClass&& move_object)
:
i{move(move_object.i)},d{move(move_object.d)}
{
    cout<<"moved";
}
void MyClass::printdata()
{
    cout<<"Integer is: "<<i<<" double is: "<<d;
}

int main()
{
    MyClass o(1,6.996);
    MyClass dak = move(o);
    dak.printdata();
}
```

```
movedInteger is: 1 double is: 6.996
Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

10. Overloading Arithmetic Operators

Write a program that overloads arithmetic operator – in terms of a compound arithmetic operator -=. Print out the values of the resulting object member fields.

```cpp
#include<iostream>
#include<string>
using namespace std;
class MyClass
{
private:
    int i;
    double d;
public:
   MyClass(int in, double dou)
   {
       i = in;
       d = dou;
   }
   MyClass& operator-=(const MyClass& rhs)
   {
       this->i -=rhs.i;
       this->d -=rhs.d;
   }
   friend MyClass operator-(MyClass lhs,const MyClass& rhs)
   {
       lhs-=rhs;
       return lhs;
   }
   void printdata()
   {
       cout<<i<< " "<<d;
   }
};


int main()
{
    MyClass a(2,4.2);
    MyClass b(1,2.2);
    MyClass result = a -b ;
    result.printdata();
}

```

```
1 2
Process returned 0 (0x0)   execution time : 0.045 s
Press any key to continue.
```