Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	1
Main Learning Points	You will learn: 1. Classes 2. Constructors and Destructors

Object Oriented Programming

Classes (I)

A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class          class_name          {
access_specifier_1:
   member1;
access_specifier_2:
   member2;
  ...
  } object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain members, that can be either data or function declarations, and optionally access specifiers.

All is very similar to the declaration on data structures, except that we can now include also functions and members, but also this new thing called *access specifier*. An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

- private members of a class are accessible only from within other members of the same class or from their *friends*.
- protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access. For example:

```
class CRectangle {
        int x, y;
      public:
        void set_values (int,int);
        int area (void);
} rect;
```

Declares a class (i.e., a type) called CRectangle and an object (i.e., a variable) of this class called rect. This class contains four members: two data members of type int (member x and member y) with private access (because private is the default access level) and two member functions with public access: set_values() and area(), of which for now we have only included their declaration, not their definition.

Notice the difference between the class name and the object name: In the previous example, CRectangle was the class name (i.e., the type), whereas rect was an object of type CRectangle. It is the same relationship int and a have in the following declaration:

```
int a;
```

where int is the type name (the class) and a is the variable name (the object).

After the previous declarations of CRectangle and rect, we can refer within the body of the program to any of the public members of the object rect as if they were normal functions or normal variables, just by putting the object's name followed by a dot (.) and then the name of the member. All very similar to what we did with plain data structures before. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that we cannot access from the body of our program outside the class are x and y, since they have private access and they can only be referred from within other members of that same class.

Here is the complete example of class CRectangle:

```
//      classes      example
#include <iostream>
 using namespace std;

class CRectangle {
       int x, y;
     public:
void set_values (int,int);
int area () {
      return (x*y);
}
};
void CRectangle::set_values (int a, int b) {
        x = a;
        y = b;
}
int main () {
     CRectangle
     rect;
     rect.set_values (3,4);
     cout << "area: " << rect.area();
   return 0;
}
```

```
area: 12
```

The most important new thing in this code is the operator of scope (::, two colons) included in the definition of set_values(). It is used to define a member of a class from outside the class definition itself.

You may notice that the definition of the member function area() has been included directly within the definition of the CRectangle class given its extreme simplicity, whereas set_values() has only its prototype declared within the class, but its definition is outside it. In this outside declaration, we must use the operator of scope (::) to specify that we are defining a function that is a member of the class CRectangle and not a regular global function.

The scope operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, in the function set_values() of the previous code, we have been able to use the variables x and y, which are private members of class CRectangle, which means they are only accessible from other members of their class.

The only difference between defining a class member function completely within its class or to include only the prototype and later its definition, is that in the first case the function will automatically be considered an inline member function by the compiler, while in the second it will be a normal (not-inline) class member function, which in fact supposes no difference in behavior.

Members x and y have private access (remember that if nothing else is said, all members of a class defined with keyword class have private access). By declaring them private we deny access to them from anywhere outside the class. This makes sense, since we have already

defined a member function to set values for those members within the object: the member function set_values(). Therefore, the rest of the program does not need to have direct access to them. Perhaps in a so simple example as this, it is difficult to see an utility in protecting those two variables, but in greater projects it may be very important that values cannot be modified in an unexpected way (unexpected from the point of view of the object).

One of the greater advantages of a class is that, as any other type, we can declare several objects of it. For example, following with the previous example of class CRectangle, we could have declared the object rectb in addition to the object rect:

```
// example: one class, two objects
#include <iostream>
 using namespace std;

class CRectangle {
        int x, y;
      public:
        void set_values (int,int);
        int area () {
return (x*y);
}
};
void CRectangle::set_values (int a, int b) {
       x = a;
       y = b;
}
int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12 rectb
area: 30
```

In this concrete case, the class (type of the objects) to which we are talking about is CRectangle, of which there are two instances or objects: rect and rectb. Each one of them has its own member variables and member functions.

Notice that the call to rect.area() does not give the same result as the call to rectb.area(). This is because each object of class CRectangle has its own variables x and y, as they, in some way, have also their own function members set_value() and area() that each uses its object's own variables to operate.

That is the basic concept of *object-oriented programming*: Data and functions are both members of the object. We no longer use sets of global variables that we pass from one function to another as parameters, but instead we handle objects that have their own data and functions embedded as members. Notice that we have not had to give any parameters in any of the calls to rect.area or rectb.area. Those member functions directly used the data members of their respective objects rect and rectb.

Constructors and destructors

1. Constructors:

Objects generally need to initialize variables or assign dynamic memory during their process of creation to become operative and to avoid returning unexpected values during their execution. For example, what would happen if in the previous example we called the member function area() before having called function set_values()? Probably we would have gotten an undetermined result since the members x and y would have never been assigned a value.
In order to avoid that, a class can include a special function called constructor, which is automatically called whenever a new object of this class is created. This constructor function must have the same name as the class, and cannot have any return type; not even void.

We are going to implement CRectangle including a constructor:

```
// example: class constructor
#include <iostream>
 using namespace std;

class CRectangle {
    int width, height;
  public:
   CRectangle (int,int);
   int area () {
        return (width*height);
}
};

CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}
int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12 rectb
area: 30
```

As you can see, the result of this example is identical to the previous one. But now we have removed the member function set_values(), and have included instead a constructor that performs a similar action: it initializes the values of x and y with the parameters that are passed to it.

Notice how these arguments are passed to the constructor at the moment at which the objects of this class are created:

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Constructors cannot be called explicitly as if they were regular member functions. They are only executed when a new object of that class is created. You can also see how neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

2. Destructors:

The *destructor* fulfils the opposite functionality. It is automatically called when an object is destroyed, either because its scope of existence has finished (for example, if it was defined as a local object within a function and the function ends) or because it is an object dynamically assigned and it is released using the operator delete.
The destructor must have the same name as the class, but preceded with a tilde sign (~) and it must also return no value. The use of destructors is especially suitable when an object assigns dynamic memory during its lifetime and at the moment of being destroyed we want to release the memory that the object was allocated.

```
// example on constructors and destructors
#include <iostream>
 using namespace std;
class CRectangle {
     int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
     int area () {
              return (*width * *height);
}
};
CRectangle::CRectangle (int a, int b) {
   width = new int;
   height = new int;
  *width = a;
  *height = b;
}
CRectangle::~CRectangle(){
        delete width;
        delete height;
}
int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12 rectb
area: 30
```

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	2
Main Learning Points	You will learn: 1. Overloading Constructors 2. Default Constructors 3. Pointers to Classes

Overloading Constructors

Like any other function, a constructor can also be overloaded with more than one function that have the same name but different types or number of parameters. Remember that for overloaded functions the compiler will call the one whose parameters match the arguments used in the function call. In the case of constructors, which are automatically called when an object is created, the one executed is the one that matches the arguments passed on the object declaration:

```
// overloading class constructors
#include <iostream>
 using namespace std;

class CRectangle {
        int width, height;
  public:
        CRectangle ();
        CRectangle (int,int);
        int area (void) {
                return (width*height);
}
};
CRectangle::CRectangle ()
{   width = 5;
    height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;   height = b;
}
int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

```
rect area: 12 rectb
area: 25
```

In this case, rectb was declared without any arguments, so it has been initialized with the constructor that has no parameters, which initializes both width and height with a value of 5.

Important: Notice how if we declare a new object and we want to use its default constructor (the one without parameters), we do not include parentheses ():

```
CRectangle    rectb;         //    right
CRectangle rectb(); // wrong!
```

Default constructor

If you do not declare any constructors in a class definition, the compiler assumes the class to have a default constructor with no arguments. Therefore, after declaring a class like this one:

```
class CExample {
   public:
       int a,b,c;
       void multiply (int n, int m) {
               a=n; b=m; c=a*b;
};
 };
```

The compiler assumes that CExample has a default constructor, so you can declare objects of this class by simply declaring them without any arguments:

```
CExample ex;
```

But as soon as you declare your own constructor for a class, the compiler no longer provides an implicit default constructor. So you have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class     CExample     {
public:
   int a,b,c;
   CExample (int n, int m) { a=n; b=m; };
void multiply () { c=a*b; };   };
```

Here we have declared a constructor that takes two parameters of type int. Therefore, the following object declaration would be correct:

```
CExample ex (2,3);
```

But,

```
CExample ex;
```

Would not be correct, since we have declared the class to have an explicit constructor, thus replacing the default constructor.

But the compiler not only creates a default constructor for you if you do not specify your own. It provides three special member functions in total that are implicitly declared if you do not

declare your own. These are the *copy constructor*, the *copy assignment operator*, and the default destructor.

The copy constructor and the copy assignment operator copy all the data contained in another object to the data members of the current object. For CExample, the copy constructor implicitly declared by the compiler would be something similar to:

```
CExample::CExample (const CExample& rv)
{   a=rv.a;  b=rv.b;  c=rv.c;
  }
```

Therefore, the two following object declarations would be correct:

```
CExample ex (2,3);
CExample ex2 (ex);   // copy constructor (data copied from ex)
```

Pointers to classes

It is perfectly valid to create pointers that point to classes. We simply have to consider that once declared, a class becomes a valid type, so we can use the class name as the type for the pointer. For example:

```
CRectangle * prect;
```

is a pointer to an object of class CRectangle.

As it happened with data structures, in order to refer directly to a member of an object pointed by a pointer we can use the arrow operator (->) of indirection. Here is an example with some possible combinations:

```
// pointer to classes example
#include <iostream>
 using namespace std;

class CRectangle {
        int width, height;
      public:
        void set_values (int, int);
        int area (void) {return (width * height);
}
};
void CRectangle::set_values (int a, int b) {
        width = a;
        height = b;
}
int main () {
  CRectangle a, *b, *c;
  CRectangle * d = new CRectangle[2];
   b= new CRectangle;
   c= &a;
  a.set_values (1,2);
  b->set_values (3,4);
   d->set_values (5,6);
  d[1].set_values (7,8);
  cout << "a area: " << a.area() << endl;
  cout << "*b area: " << b->area() << endl;
  cout << "*c area: " << c->area() << endl;
 cout << "d[0] area: " << d[0].area() << endl;
   cout << "d[1] area: " << d[1].area() << endl;
   delete[] d;   delete b;   return 0;
}
```

```
a  area:  2  *b
area:   12   *c
area:  2  d[0]
area: 30 d[1]
area: 56
```

Next you have a summary on how can you read some pointer and class operators (*, &, ., ->, [ ]) that appear in the previous example:

expression	can be read as
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x

Classes defined with struct and union

Classes can be defined not only with keyword class, but also with keywords struct and union. The concepts of class and data structure are so similar that both keywords (struct and class) can be used in C++ to declare classes (i.e. structs can also have function members in C++, not only data members). The only difference between both is that members of classes declared with the keyword struct have public access by default, while members of classes declared with the keyword class have private access. For all other purposes both keywords are equivalent.

The concept of unions is different from that of classes declared with struct and class, since unions only store one data member at a time, but nevertheless they are also classes and can thus also hold function members. The default access in union classes is public.

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	3
Main Learning Points	You will learn: 1. Overloading Operators

Classes (II)

Overloading operators

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

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

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

```
struct {
string product;
float price;
} a, b, c;
 a = b + c;
```

In fact, this will cause a compilation error, since we have not defined the behavior our class should have with addition operations. However, C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

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

To overload an operator in order to use it with classes we declare *operator functions*, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload. The format is:

type operator sign (parameters) { /*...*/ }

Here you have an example that overloads the addition operator (+). We are going to create a class to store bidimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bidimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1,1+2) = (4,3).

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

class CVector {
     public:     int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

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

CVector CVector::operator+ (CVector param) {
   CVector temp;
   temp.x = x + param.x;
   temp.y = y + param.y;
   return (temp);
}
int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;   return 0;
}
```

```
4,3
```

It may be a little confusing to see so many times the CVector identifier. But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

```
CVector (int, int);            // function name CVector (constructor)
CVector operator+ (CVector);   // function returns a CVector
```

The function operator+ of class CVector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:

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

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

```
CVector () { };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

```
CVector c;
```

included in main() would not have been valid.

Anyway, an empty block is a bad implementation for a constructor, since it does not fulfill the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

```
CVector () { x=0; y=0; };
```

which in order to simplify and show only the point of the code I have not included in the example.

As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (=) with the class itself as parameter. The behavior which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
CVector d (2,3);
CVector e;
```

```
e = d;          // copy assignment operator
```

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use operator + to subtract two classes or operator== to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function operator+ can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

<table>
<tr><th>Expression</th><th>Operator</th><th>Member function</th><th>Global function</th></tr>
<tr><td>@a</td><td>+ - * & ! ~ ++ --</td><td>A::operator@()</td><td>operator@(A)</td></tr>
<tr><td>a@</td><td>++ --</td><td>A::operator@(int)</td><td>operator@(A,int)</td></tr>
<tr><td>a@b</td><td>+ - * / % ^ & | < > == != <= >= << >> && || ,</td><td>A::operator@ (B)</td><td>operator@(A,B)</td></tr>
<tr><td>a@b</td><td>= += -= *= /= %= ^= &= |= <<= >>= []</td><td>A::operator@ (B)</td><td>-</td></tr>
<tr><td>a(b, c...)</td><td>()</td><td>A::operator() (B, C...)</td><td>-</td></tr>
<tr><td>a->x</td><td>-></td><td>A::operator->()</td><td>-</td></tr>
</table>

Where a is an object of class A, b is an object of class B and c is an object of class C.

You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	4
Main Learning Points	You will learn: 1. Keyword 'This' 2. Static members

The keyword this

The keyword this represents a pointer to the object whose member function is being executed. It is a pointer to the object itself.

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

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

class CDummy {
  public:    int isitme (CDummy& param);

};
int CDummy::isitme (CDummy& param)
{
  if (&param == this)
           return true;
   else
          return false;
}
int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
     cout << "yes, &a is b";
  return 0;
}
```

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

It is also frequently used in operator= member functions that return objects by reference (avoiding the use of temporary objects). Following with the vector's examples seen before we could have written an operator= function similar to this one:

```
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 if we do not include an operator= member function to copy objects of this class.

Static members

A class can contain *static* members, either data or functions.

Static data members of a class are also known as "class variables", because there is only one unique value for all the objects of that same class. Their content is not different from one object of this class to another.

For example, 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:

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

class CDummy {
  public:     static int n;
            CDummy () {
            n++; };
            ~CDummy () { n--; };
};
int CDummy::n=0;

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

```
7
6
```

In fact, static members have the same properties as global variables but they enjoy class scope. For that reason, and to avoid them to be declared several times, we can only include the prototype (its declaration) in the class declaration but not its definition (its initialization). In order to initialize a static data-member we must include a formal definition outside the class, in the global scope, as in the previous example:

```
int CDummy::n=0;
```

Because it is a unique 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):

```
cout << a.n;
cout << CDummy::n;
```

These two calls included in the previous example are referring to the same variable: the static variable n within class CDummy shared by all objects of this class.

Once again, in fact it is a global variable. The only difference is its name and possible access restrictions outside its class.

Just as we may include static data within a class, we can also include static functions. They represent the same: they are global functions that are called as if they were object members of a given class. They can only refer to static data, in no case to non-static members of the class, as well as they do not allow the use of the keyword this, since it makes reference to an object pointer and these functions in fact are not members of any object but direct members of the class.

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	5
Main Learning Points	You will learn: 1. Friend function 2. Friend Class

Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.
Friends are functions or classes declared as such.
If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
// friend functions #include <iostream>
using namespace std;
class CRectangle {
  int width, height;
 public:
  void set_values (int, int);
   int area () {return (width * height);}
   friend CRectangle duplicate (CRectangle);
};
void CRectangle::set_values (int a, int b) {
  width = a;   height = b;
}
 CRectangle duplicate (CRectangle rectparam)
{
 CRectangle rectres;
 rectres.width = rectparam.width*2;
 rectres.height = rectparam.height*2;
 return (rectres);
}
int main () {
 CRectangle rect, rectb;
 rect.set_values (2,3);
 rectb = duplicate (rect);
 cout << rectb.area();
 return 0;
}
```

```
24
```

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

Friend classes

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
// friend class
#include <iostream>
using namespace std;

class CSquare;
class CRectangle {
int width, height;
  public:
      int area ()
        {return (width * height);}
      void convert (CSquare a);
};
class CSquare {
   private:
       int side;
   public:
       void set_side (int a)
            {side=a;}
       friend class CRectangle;
};
void CRectangle::convert (CSquare a) {
width = a.side;   height = a.side;
}
int main () {
CSquare sqr;
CRectangle rect;
sqr.set_side(4);
rect.convert(sqr);
cout << rect.area();
return 0;
}
```

```
16
```

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle.

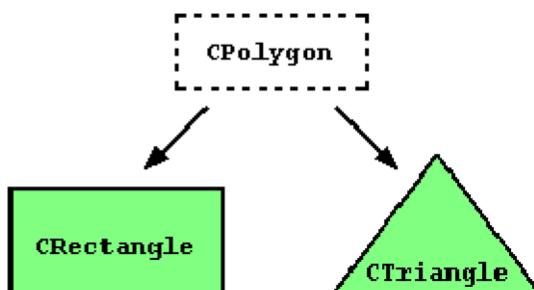
Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to.

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	6
Main Learning Points	You will learn: 1. Inheritance

Inheritance between classes

A key feature of C++ classes is inheritance. Inheritance allows to create classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own. For example, we are going to suppose that we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base.

This could be represented in the world of classes with a class CPolygon from which we would derive the two other ones: CRectangle and CTriangle.



The class CPolygon would contain members that are common for both types of polygon. In our case: width and height. And CRectangle and CTriangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive it to another class with another member called B, the derived class will contain both members A and B.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name:
public base_class_name { /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier describes the minimum access level for the members that are inherited from the base class.

```
// derived classes
#include <iostream>
using namespace std;

class CPolygon {
  protected:
      int width, height;
  public:
      void set_values (int a, int b)
          { width=a; height=b;}
  };
class CRectangle:
 public CPolygon {
     public:
         int area ()
          { return (width * height); }
  };
class CTriangle:
  public CPolygon {
         public:
            int area ()
                { return (width * height / 2); }
  };
int main () {
  CRectangle rect;
  CTriangle trgl;
   rect.set_values (4,5);
   trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

```
20
10
```

The objects of the classes CRectangle and CTriangle each contain members inherited from CPolygon. These are: width, height and set_values().

The protected access specifier is similar to private. Its only difference occurs in fact with inheritance. When a class inherits from another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used protected access instead of private.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

Where "not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

```
CPolygon::width          // protected access
CRectangle::width        // protected access

CPolygon::set_values()   // public access
CRectangle::set_values() // public access
```

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes:

```
class CRectangle: public CPolygon { ... }
```

This public keyword after the colon (:) denotes the maximum access level for all the members inherited from the class that follows it (in this case CPolygon). Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

If we specify a more restrictive access level like protected, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: private, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

```
class daughter: protected mother;
```

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

If we do not explicitly specify any access level for the inheritance, the compiler assumes private for classes declared with class keyword and public for those declared with struct.

What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are always called when a new object of a derived class is created or destroyed.

If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:
derived_constructor_name (parameters) : base_constructor_name (parameters) {...}
For example:

```
// constructors and derived classes
#include <iostream>
using namespace std;

class mother {
  public: mother () {
      cout << "mother: no parameters\n"; }
      mother (int a) {
        cout << "mother: int parameter\n"; }
};
class daughter : public mother {
public:     daughter (int a) {
         cout << "daughter: int parameter\n\n"; }
};
class son : public mother {
    public:
      son (int a) : mother (a)
     { cout << "son: int parameter\n\n"; }
};
int main () {   daughter
cynthia (0);   son
daniel(0);

  return 0;
}
```

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object. The difference is because the constructor declaration of daughter and son:

```
daughter (int a)          // nothing specified: call default
son (int a) : mother (a)  // constructor specified: call this
```

Multiple inheritance

In C++ it is perfectly possible that a class inherits members from more than one class. This is done by simply separating the different base classes with commas in the derived class declaration. For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write:

```
class CRectangle: public CPolygon, public
COutput; class CTriangle: public CPolygon,
public COutput;
```

here is the complete example:

```
// multiple inheritance #include
<iostream>
using namespace std;

class CPolygon {
   protected:
      int width, height;
   public:
      void set_values (int a, int b)
        { width=a; height=b;}
 };
class COutput {
    public:
      void output (int i);
 };
 void COutput::output (int i) {
      cout << i << endl;
 }
class CRectangle: public CPolygon, public COutput {
     public:
         int area ()
           { return (width * height); }
 };
class CTriangle: public CPolygon, public COutput {
     public:
        int area ()
          { return (width * height / 2); }
 };
int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
```

```
20
10
```

```
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	7
Main Learning Points	You will learn: 1. Polymorphism 2. Pointers to Base Class 3. Virtual Members

Pointers to base class
One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential. We are going to start by rewriting our program about the rectangle and the triangle taking into consideration this pointer compatibility property:

```
// pointers to base class
#include <iostream>
using namespace std;
class CPolygon {
protected:
   int width, height;
public:
 void set_values (int a, int b)
     { width=a; height=b; } };
class CRectangle:  public CPolygon {
 public:
      int area ()
     { return (width * height); } };
class CTriangle: public CPolygon {
public:
   int area ()
     { return (width * height / 2); } };
int main () {
CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
cout << rect.area() << endl;
cout << trgl.area() << endl;
return 0;
}
```

```
20
10
```

In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations.
The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.
In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:

Virtual members
A member of a class that can be redefined in its derived classes is known as a virtual member. In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual:

```
// virtual members
#include <iostream>
using namespace std;
class CPolygon {
protected:
   int width, height;
public: void set_values (int a, int b) {
width=a; height=b; }
virtual int area ()
     { return (0); } };
class CRectangle: public
CPolygon {
public:    int area ()
     { return (width * height); } };
class CTriangle: public CPolygon {
public:     int area ()
     { return (width * height / 2); } };
int main () {
CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
CPolygon * ppoly3 = &poly;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly3->set_values (4,5);
cout << ppoly1->area() << endl;
cout << ppoly2->area() << endl;
cout << ppoly3->area() << endl;
return 0;
}
```

```
20
10
0
```

Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the previous example.

A class that declares or inherits a virtual function is called a *polymorphic class*.

Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to call its own area() function, which always returns 0.

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	8
Main Learning Points	You will learn: 1. Abstract Base Classes

Abstract base classes

Abstract base classes are something very similar to our CPolygon class of our previous example. The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an abstract base classes we could leave that area() member function without implementation at all. This is done by appending =0 (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:

```
// abstract class CPolygon
class CPolygon {
protected:
   int width, height;
 public:
void set_values (int a, int b)
{ width=a; height=b; }
virtual int area () =0;
};
```

Notice how we appended =0 to virtual int area () instead of specifying an implementation for the function. This type of function is called a *pure virtual function*, and all classes that contain at least one pure virtual function are *abstract base classes*.

The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.
But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore, a declaration like:

```
CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:

```
CPolygon * ppoly1;
CPolygon * ppoly2;
```

would be perfectly valid.
This is so for as long as CPolygon includes a pure virtual function and therefore it's an abstract base class. However, pointers to this abstract base class can be used to point to objects of derived classes.
Here you have the complete example:

```
// abstract base class
#include <iostream>
using namespace std;
class CPolygon {
protected:
  int width, height;   public:
void set_values (int a, int b) {
width=a; height=b; }
 virtual int area (void) =0; };
class CRectangle: public CPolygon {
public:
 int area (void)
     { return (width * height); }  };
class CTriangle: public CPolygon {
public:
   int area (void)
     { return (width * height / 2); }
  };
int main () {
CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
 cout << ppoly1->area() << endl;
cout << ppoly2->area() << endl;
return 0;
}
```

```
20
10
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;
class CPolygon {
protected:
  int width, height;
public:
  void set_values (int a, int b) {
    width=a; height=b; }
virtual int area (void) =0;
void printarea (void)
     { cout << this->area() << endl; }
  };
class CRectangle: public CPolygon {
public:     int area (void)
     { return (width * height); }
  };
class CTriangle: public CPolygon {
public:     int area (void)
     { return (width * height / 2); }
  };
int main () {
CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly1->printarea();
ppoly2->printarea();
return 0;
}
```

```
20
10
```

Virtual members and abstract classes grant C++ the polymorphic characteristics that make object-oriented programming such a useful instrument in big projects. Of course, we have seen very simple uses of these features, but these features can be applied to arrays of objects or dynamically allocated objects.

Let's end with the same example again, but this time with objects that are dynamically allocated:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon {
protected:
  int width, height;
public:
  void set_values (int a, int b)
  { width=a; height=b; }
  virtual int area (void) =0;
  void printarea (void)
    { cout << this->area() << endl; }
 };
class CRectangle: public CPolygon {
public:    int area (void)
    { return (width * height); }
 };
class CTriangle: public CPolygon {
public:    int area (void)
    { return (width * height / 2); }
 };
int main () {
 CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly1->printarea();
ppoly2->printarea();
delete ppoly1;
delete ppoly2;
return 0;
}
```

```
20
10
```

Notice that the ppoly pointers:

```
CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;
```

are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	9
Main Learning Points	You will learn: 1. Function Templates 2. Class Templates

Templates

Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type. In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way. For example, to create a template function that returns the greater one of two objects we could use:

```
template <class myType> myType GetMax (myType a, myType b)
{
 return (a>b?a:b);
}
```

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. The function template GetMax returns the greater of two parameters of this still undefined type.

To use this function template, we use the following format for the function call:

```
function_name <type> (parameters);
```

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

```
//    function    template
#include      <iostream>
using namespace std;

template <class T> T GetMax (T a, T b)
{
 T result;
 result = (a>b)? a : b;
 return (result);
}
int main () {
   int i=5, j=6, k;
   long l=10, m=5, n;
   k=GetMax<int>(i,j);
   n=GetMax<long>(l,m);
  cout << k << endl;
  cout << n << endl;
  return 0;
}
```

```
6
10
```

In this case, we have used T as the template parameter name instead of myType because it is shorter and in fact is a very common template parameter name. But you can use any identifier you like.

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The compiler has instantiated and then called each time the appropriate version of the function. The type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

In this specific case where the generic type T is used as a parameter for GetMax the compiler can find out automatically which data type has to instantiate without having to explicitly specify it within angle brackets (like we have done before specifying <int> and <long>). So we could have written instead:

```
int i,j; GetMax (i,j);
```

Since both i and j are of type int, and the compiler can automatically find out that the template parameter can only be int. This implicit method produces exactly the same result:

```
// function template II #include <iostream>
using namespace std;

template <class T> T GetMax (T a, T b) {
  return (a>b?a:b);
}
int main () {
   int i=5, j=6, k;
   long l=10, m=5, n;
   k=GetMax(i,j);
   n=GetMax(l,m);
   cout << k << endl;
   cout << n << endl;
   return 0;
}
```

```
6
10
```

Notice how in this case, we called our function template GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one template parameter (class T) and the function template itself accepts two parameters, both of this T type, we cannot call our function template with two objects of different types as arguments:

```
int i; long l;
k = GetMax (i,l);
```

This would not be correct, since our GetMax function template expects two arguments of the same type, and in this call to it we use objects of two different types.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U> T GetMin (T a, U b) {
  return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j; long l;
i = GetMin<int,long> (j,l);
```

or simply:

```
i = GetMin (j,l);
```

even though j and l have different types, since the compiler can determine the appropriate instantiation anyway.

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T> class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first;
      values[1]=second;
    }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

The only member function in the previous class template has been defined inline within the class declaration itself. In case that we define a function member outside the declaration of the class template, we must always precede that definition with the template <...> prefix:

```
// class templates #include
<iostream> using namespace std;

template <class T> class mypair {
    T a, b;
  public:
    mypair (T first, T second)
     {a=first; b=second;}
    T getmax ();
};
template <class T> T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}
int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

```
100
```

Notice the syntax of the definition of member function getmax:

```
template <class T> T mypair<T>::getmax ()
```

There are three T's in this declaration: The first one is the template parameter. The second T refers to the type returned by the function. And the third T (the one between angle brackets) is also a requirement: It specifies that this function's template parameter is also the class template parameter.

Document Information	
Course	Object Oriented Programming (C++)
Lecture Number	10
Main Learning Points	You will learn: 1. Template Specialization 2. Templates and multiple-file projects

Template specialization

If we want to define a different implementation for a template when a specific type is passed as template parameter, we can declare a specialization of that template.

For example, let's suppose that we have a very simple class called mycontainer that can store one element of any type and that it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, so we decide to declare a class template specialization for that type:

```
// template specialization
#include <iostream>
using namespace std;

// class template:
template <class T> class mycontainer {
      T element;
   public:
      mycontainer (T arg)
         {element=arg;}
      T increase () {return ++element;}
};

// class template specialization:
template <> class mycontainer <char> {
      char element;
   public:
      mycontainer (char arg)
        {element=arg;}
      char uppercase () {
     if((element>='a')&&(element<='z')) element+='A'-'a';
      return element;
   }
};
int main () {
  mycontainer<int> myint (7);
```

```
8
J
```

```
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
  return 0;
}
```

This is the syntax used in the class template specialization:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an emptytemplate<> parameter list. This is to explicitly declare it as a template specialization.

But more important than this prefix, is the <char> specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (char). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
 template <> class mycontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization.

When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Non-type parameters for templates

Besides the template arguments that are preceded by the class or typename keywords , which represent types, templates can also have regular typed parameters, similar to those found in functions. As an example, have a look at this class template that is used to contain sequences of elements:

```
// sequence template
#include <iostream>
using namespace std;

template <class T, int N> class mysequence {
    T memblock [N];
  public:
    void setmember (int x, T value);
    T getmember (int x);
};
template <class T, int N>
void mysequence<T,N>::setmember (int x, T value) {
  memblock[x]=value;
}
template <class T, int N>
T mysequence<T,N>::getmember (int x) {
  return memblock[x];
}
int main () {
   mysequence <int,5> myints;
   mysequence <double,5> myfloats;
   myints.setmember (0,100);
   myfloats.setmember (3,3.1416);
   cout << myints.getmember(0) << '\n';
   cout << myfloats.getmember(3) << '\n';
  return 0;
}
```

```
100
3.1416
```

It is also possible to set default values or types for class template parameters. For example, if the previous class template definition had been:

```
template <class T=char, int N=10> class mysequence {..};
```

We could create objects using the default template parameters by declaring:

```
mysequence<> myseq;
```

Which would be equivalent to:

```
mysequence<char,10> myseq;
```

Templates and multiple-file projects

From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation with specific template arguments is required. At that moment, when an

instantiation is required, the compiler generates a function specifically for those arguments from the template.

When projects grow it is usual to split the code of a program in different source code files. In these cases, the interface and implementation are generally separated. Taking a library of functions as example, the interface generally consists of declarations of the prototypes of all the functions that can be called. These are generally declared in a "header file" with a .h extension, and the implementation (the definition of these functions) is in an independent file with c++ code.

Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates.

Since no code is generated until a template is instantiated when required, compilers are prepared to allow the inclusion more than once of the same template file with both declarations and definitions in a project without generating linkage errors.