Software Engineering
C++ and Object Oriented Programming

Yohan FOUGEROLLE,
Département Ge2i, Bureau 005,

Centre Universitaire Condorcet
Laboratoire Le2i - UMR CNRS 5158
12 rue de la Fonderie - 71200 LE CREUSOT

Tel : 03 85 73 11 37 ✉ yohan.fougerolle@u-bourgogne.fr

Outline

Syllabus

- **Fundamentals of Oriented Object Programming**
  - **Classes and objects**
  - **Pointers to classes**
  - **Overloading operators**
  - **Static members and functions**
  - **Constructors and destructors**
- **Friendship and inheritance**
- **Polymorphism**
- **Template functions and template classes**

Classes

- 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. Syntax:

```
class class_name {
 access_specifier_1: // can be public, protected or private
  type1 member1;
  …
 access_specifier_2: // same
  type2 member2;
  …
} object_names; // you can instantiate objets after class declaration
```

- `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.

Example1

```
class CRectangle { //class declaration

private:
    int width, height;          // two private attribute members of type int

public:
     Crectangle ( );            // default constructor
     CRectangle ( int, int);    // constr w. Params
     CRectangle ( CRectangle &); // copy constr.
    ~CRectangle ( );            // destructor
    int area () { return (width * height); } // a public function member
};

//Definitions for member functions
CRectangle :: CRectangle ( ) {  width = height = 0; }
CRectangle :: CRectangle (int a, int b) { width = a; height = b; }
CRectangle :: CRectangle ( const CRectangle &R ) {
      width = R.width; height = R.height;
}
CRectangle :: ~CRectangle ( ) { }
```

Example 2

```
class CRectangle {

private:
    int *width, *height;        // WATCH OUT HERE! int* and no longer int

public:

    CRectangle ( );             // default constructor
    CRectangle ( int, int);     // constr w. params
   ~CRectangle ( );             // destructor
    int area () { return (*width * *height); }
};

CRectangle :: CRectangle ( width = NULL; height = NULL;}

CRectangle :: CRectangle (int a, int b) {
   width = new int;  height = new int; // dynamic allocation of integers
  *width = a;       *height = b;
}

Crectangle :: CRectangle (const CRectangle &R) {
   width = new int;  height = new int; // dynamic allocation of integers
  *width = R.width; *height = R.height;
}

//member allocated dynamically MUST be deteleted with delete operator:

CRectangle :: ~CRectangle () {
  delete width;
  delete height;
}
```

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

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

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

The operator new

For classes, the operator new calls the constructors (default, parameters, copy, etc) and returns a pointer to the object created :

```
CRectangle *p_rect_a, *p_rect_b;  //pointers declarations

// call to the constructors using new
p_rect_a = new CRectangle;        //default constructor call
p_rect_b = new CRectangle (7,8);  //parameters constructor

// objects are created, we can use them through their pointers
cout << "rect_a area: " << p_rect_a->area() << endl;
cout << "rect_b area: " << p_rect_b->area() << endl;

// suppose we have finished, now we free the memory
delete p_rect_a;
delete p_rect_b;
```

```
// overloading class constructors

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

//default constructor
CRectangle::CRectangle () { width = 5;  height = 5;}

// param constr.
CRectangle::CRectangle (int a, int b) { width = a;  height = b;}

int main ( ) {

  CRectangle rect_a (3,4);   // constr. params.
  CRectangle rect_b;         // default constr for static obj
  CRectangle *rect_c = new CRectangle(5,6); //cstr. para.
  CRectangle *rect_d = new CRectangle;      // default too

  cout << "rect_a area: " << rect_a.area() << endl;  //12
  cout << "rect_b area: " << rect_b.area() << endl;  //25
  cout << "rect_a area: " << rect_c->area() << endl; //30
  cout << "rect_b area: " << rect_d->area() << endl; //25
}
```

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. **Advice: always provide a default constructor.** After declaring a class like :

```
class CExample { // class without constructor implemented…
  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. You have to declare all objects of that class according to the constructor prototypes you defined for the class:

```
class CExample2 { //class with only param. cstr implemented…
  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.

```
CExample ex (2,3); //correct
CExample ex;        //incorrect
```

Constructors and destructors

- **Constructors initialize** the values (**and allocate** memory, if needed)

- Constructors **cannot be called explicitly** as if they were regular member functions (only executed when a new object of that class is created)

- Neither the constructor prototype declaration (within the class) nor the latter constructor definition include a return value; not even void.

- The **destructor fulfills the opposite functionality** and is automatically called when an object is **destroyed**. This happens in two cases :
  - because its **scope of existence has finished** (for example, if it was defined as a local object within a function and the function ends)
  - 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**.

```
class CRectangle {
// watch out: we use pointers for this example to illustrate the need of dedicated code
// in the destructor...
  private:
    int *width, *height;  // pointers now!
  public:
    // no default constructor, it is bad but it is allowed...
     CRectangle (int, int); // cons. param.
    ~CRectangle ( ) ;       // destructor
    int area ( ) { return (*width * *height); }
};

CRectangle :: CRectangle (int a, int b) {
   //dynamic allocation when creating an object
   width = new int; height = new int;
  *width = a;  *height = b; //assignment
}

CRectangle :: ~CRectangle ( ) {
  // free memory dynamically allocated
  delete width; delete height;
}

int main ( ) {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;     // 12
  cout << "rectb area: " << rectb.area() << endl;   // 30
  return 0;
} < - - - destructors are called here : the scope of existence of both objects ends here
```

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 * p_rect; //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.
- Next you have a summary on how can you read some pointer and class:

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

The `this` pointer

- `this` is an implicit parameter to all the member functions of an object and is a **pointer to the object itself**
- Useful when a function which updates an object must return the object after modification
- Very useful to observe which objects call a function

```
// example on constructors and destructors with pointers
#include <iostream>
using namespace std;

class CRectangle { //same class
  private:
       int width, height;
  public:
       CRectangle (int, int);
       ~CRectangle ( ); // destructor
       int area () { return (width * height); }
};

CRectangle::CRectangle (int a, int b) {
  width = a;  height = b;
}

CRectangle::~CRectangle () {
  //nothing to do here since width and height
  // are located int... the stack (not the heap like for ptrs)
  // automatically erased when object is erased
}

int main () {

  CRectangle rect (3,4);
  CRectangle *p_rect = &rect; // p_rect is a pointer to rect
  cout << "rect area: " << rect.area() << endl;     // 12
  cout << "rectb area: " << p_rect->area() << endl; // 12
  return 0;
}
```

Overloading operators

- 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.
- Syntax:

```
<type> operator <sign> ( <parameters> ) { /*...*/ }
```

Overloadable operators

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

- Next 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 ( ) { };    // default constructor
    CVector (int, int); // param. constructor
    CVector operator + (const CVector&); // overload of operator +
};

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

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

int main () {
  CVector a (3,1), b (1,2);
  CVector c;   c = a + b
  cout << c.x << "," << c.y; // 4,3
  return 0;
}
```

Overloading operators

- 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.
  - You can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures
  - To be safe, systematically overload it yourself

Overloading operators

- The prototype of a function operator+ can seem obvious : 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.
- Below, you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

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

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. Example:

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

class CDummy {
        public:
        //declaration of a common member to all the instances of Cdummy class
        static int n;
        //to illustrate the interest of such shared variable:
        CDummy ( ) { n++; };
        ~CDummy ( ) { n--; };
};

// initialisation can be done anywhere… (who said global variable?)
int CDummy :: n = 0; // note the usage of scope operator ::

int main ( ) {
  CDummy a;                          // 1 allocation --> n == 1
  CDummy b[5];                       // 5 allocations --> n == 6
  CDummy * c = new CDummy;           // 1 more allocation --> n == 7
  cout << a.n << endl;               // 7
  delete c;                          // 1 deletion --> n == 6
  cout << CDummy::n << endl;         // 6
  return 0;
}
```

Static members

- In fact, static members have the similar properties as global variables but with class scope (operator ::)
- 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; // global initialisation
```

- 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
  - this is only valid for static members

```
cout << a.n; // correct but not really elegant since n is a static member
cout << CDummy :: n;  // much better !
```

Static members

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

- Once again, I remind you that is a global variable.
  - The only difference are
    - its name
    - 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.
  - Sometimes referred as “comfort” functions : functions which are “comfortable” to use since they do not require any object creation
  - They can only refer to static data, in no case to non-static members of the class
  - 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

Access specifiers

- These specifiers modify the access rights that the members/functions acquire:
  - **private:** members of a class are **accessible only from within other members of the same class** or **from their friends**.
  - **protected:** same as private, but also from members of their **derived classes**
  - **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. For example:

```
class CRectangle {
  int x, y; // default specifier is private
  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)
- two member functions with public access: `set_values()` and `area()`

Classes

- Notice the **difference** between the **class name** and the **object name**
  - In the previous example, `CRectangle` was the class name (i.e., the type)
  - **`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. 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.

```
// access specifier example

#include <iostream>
using namespace std;

class CRectangle {

    int x, y; // private by default

  public:

    void set_values (int, int); // public access
    int area ( ) { return (x*y); } // public access
};

void CRectangle :: set_values (int a, int b) {  x = a;  y = b; }

int main ( ) {

  CRectangle rect;

  rect.set_values (3,4); // call ok : set_value is public
  cout << "area: " << rect.area(); // same

  cout << rect.x << endl; //error : main cannot access member x

  return 0;
}
```

Explanations

- 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.** Two Remarks :
  - The definition of the member function `area( )` has been included directly within the definition of the `CRectangle` class given its extreme simplicity.
  - `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 `set_values()`, we have been able to use the variables x and y, which are private members of class `CRectangle :` this 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
  - in the first case, the function will automatically be considered an inline member function
  - in the second it will be a normal (not-inline) class member function

Explanations

- Members x and y have private access :
  - 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**.
  - We have already defined a member function to set values for those members within the object: 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 any utility in protecting those two variables
  - 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).
  - Think about the bank account example...

Up to now, what do we know?

- Basics in Object Oriented Programming
  - Constructor(s) : default, with parameters, copy constructor, etc
  - Destructor
  - Member functions and attributes (this, etc)
  - Access rights : private, public, and protected

- Now :
  - Classes can be derived to create new classes (inheritance)
  - Classes and functions can be declared as friends to other classes to be granted access to member functions and attributes
  - Polymorphism...

Friendship and inheritance

- friend functions
  - `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 classes and friend functions
  - 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:
    - Do it by declaring a prototype of this external function within the class
    - Precede it with the keyword `friend`

- Friend classes
  - Just as we have the possibility to define a friend function, we can also define a class as friend of another one
  - This means that the first class is granted access to the protected and private members of the second one

```cpp
#include <iostream>
using namespace std;

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

       // friend function declaration
       friend CRectangle duplicate (const CRectangle&);
};

void CRectangle :: set_values (int a, int b) { width = a;  height = b; }

CRectangle duplicate (const CRectangle& rectparam) {
// look : this function is not a member function (no CRectangle ::)
// Nevertheless, it can access the width and height members of CRectangle
  CRectangle result;
  result.width = rectparam.width*2; // access to width for both objects
  result.height = rectparam.height*2; // access to height for both objects
  return (result);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3); cout << rect.area();       // 6
  rectb = duplicate (rect);  cout << rectb.area();  //24
  return 0;
}
```

Explanations

- 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()` as a member of class `CRectangle.` 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`

```
#include <iostream>
using namespace std;
class CSquare; // why is this here?

class CRectangle {
    int width, height; // 2 private attributes
  public:
    int area ()      { return (width * height); }
    void convert ( const CSquare& );
};

class CSquare {
    int side; // only 1 private attribute
  public:
    void set_side (int a)      { side =a; }
    friend class CRectangle; // what does it mean?
};

void CRectangle :: convert (const CSquare& a) {
  width = a.side;  height = a.side; //access to CSquare :: side
}

int main ( ) {

  CSquare sqr;   CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area(); // 16
  return 0;
}
```

Explanations

- We have declared **CRectangle as a friend of CSquare**
  - CRectangle member functions could have access to the protected and private members of CSquare
  - more concretely, to CSquare :: side for this example

- 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: if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle

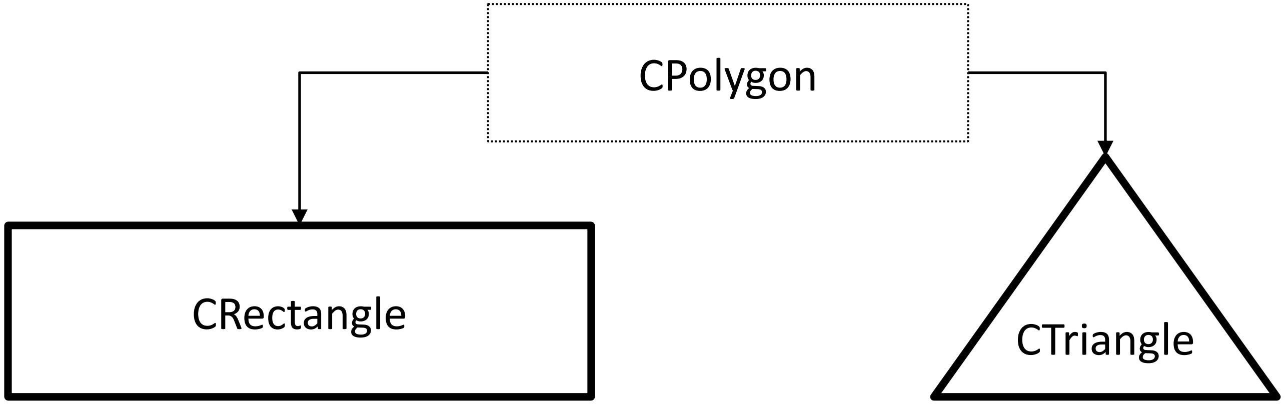
- Consider that friendships are **NOT bidirectional** if we do not explicitly specify so
  - In our example, CRectangle is considered as a friend class by CSquare
  - CRectangle does not consider CSquare to be a friend
  - CRectangle can access the protected and private members of CSquare but not the reverse way

- Another property of friendships is that they are **not transitive**
  - The friend of a friend is not considered to be a friend

Inheritance between classes

- 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 `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`



Inheritance between classes

- The class `CPolygon` would contain members that are common for both types of polygons. In our case: `width` and `height`
- `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
{ /*...*/ };
```

- The `public` access specifier may be replaced by any one of the other access specifiers **`protected`** or **`private`**. This access specifier limits the **most accessible level** for the members inherited from the base class: the members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

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

class CPolygon {
  protected: //to allow transmission to derived classes
    int width, height;
  public:
    void set_values (int a, int b)      { width=a; height=b; }
  };

class CRectangle: public CPolygon { // CRectangle inherits from CPolygon
  public:
    int area ()      { return (width * height); } // member function added
  };

class CTriangle: public CPolygon { // CTriangle inherits from CPolygon
  public:
    int area ()      { return (width * height / 2); } // member function added
  };

int main ( ) {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5); //call to set value, defined in CPolygon
  trgl.set_values (4,5); //call to set value, defined in CPolygon
  cout << rect.area() << endl; //20
  cout << trgl.area() << endl; //10
  return 0;
}
```

Explanations

- The objects of the classes `CRectangle` and `CTriangle` each contain members inherited from `CPolygon`
  - `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.

- The following tables summarize the different access types according to all the inheritance modes

Inheritance mode

Inheritance type	Access specifier in the mother class	Access specifier in the derived class
public	public protected private	public protected private **(not accessible)**
protected	public protected private	protected protected private **(not accessible)**
private	public protected private	private **(accessible)** private **(accessible)** private **(not accessible)**

Accessibility ?	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` because we have used the `public` keyword to define the inheritance relationship on each of the derived classes:

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

- In other words :

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

What is inherited from the base class?

- **A derived class inherits every member of a base class except**:
  - its constructor and its destructor
  - its operator =
  - its friends

- Although the constructors and destructors of the base class are not inherited themselves, **the base class default constructor and 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 if 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 (params) : base_constructor_name (params) {/*...*/}
```

Main program (constructors and derived classes)

```cpp
#include <iostream>
using namespace std;

class mother {
  public: // declare dummy constructor for display
    mother ( )    { cout << "mother: no parameters\n"; }
    mother (int a){ cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public: // no default constructor so...
    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); // creates a daughter… which constr??
  son daniel(0);        // creates a son… which constr??

  return 0;
}
```

Result

```
mother: no parameters
daughter: int
parameter

mother: int parameter
son: int parameter
```

Explanations

- 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
son (int a) : mother (a)  // call to param. mother constructor specified
```

- Another example:

```
class X {
public :
    int a, b;
    //default constructor
    X ( ) { a=0; b=0; }
    //param constructor
    X (int i, int j ) { a=i; b=j; }
…
};
```

```
class Y : public X { //public inheritance for simplicity
public :
    int c; // we add a member
     //default constructor
    Y ( ) : X( ), c(0) { /*some code*/ }
     //param constructor
    Y (int aa, int bb, int cc = 0) : X(aa,bb), c(cc) {…}
… };
```

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;
class CPolygon {
  protected:    int width, height; // to allow transmission to inherited classes
  public:    void set_values (int a, int b) { width=a; height=b;}
  };

class COutput {  //dummy class which can only use its output function
  public:
  void output ( int i ) {  cout << i << endl;   }
};

class CRectangle: public CPolygon, public COutput { //multiple inheritance
  public:
  int area ( )      { return (width * height); } // we add an area function
  };

class CTriangle: public CPolygon, public COutput { //multiple inheritance
  public:
  int area ( )      { return (width * height / 2); } // we add another area function
  };

int main ( ) {
       CRectangle rect;   rect.set_values (4,5);
       CTriangle trgl;  trgl.set_values (4,5);

       // call the output( ) function inherited from COutput
       // with, as argument, the result of a call to the area( ) function
         rect.output (rect.area());  // 20
         trgl.output (trgl.area());  // 10
         return 0;
}
```

Polymorphism

- 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.** We are going to start by rewriting our program about the rectangle and the triangle, taking into consideration this pointer compatibility property

- In function main, **we create two pointers** that point to objects of class **`CPolygon`** : `ppoly1` and `ppoly2`

- **We assign references to `rect` and `trgl` to these pointers.** 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***
  - 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

- 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

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

class CRectangle: public CPolygon { //as usual
  public:
    int area ( )      { return (width * height); }
  };

class CTriangle: public CPolygon {  //as usual
  public:
    int area ( )      { return (width * height / 2); }
  };

int main () {

  CRectangle rect;
  CTriangle trgl;

  CPolygon * ppoly1 = &rect;// look out : a ptr to Polygon = address of a CRect
  CPolygon * ppoly2 = &trgl;// look out : a ptr to Polygon = address of a CTrgl

  ppoly1->set_values (4,5); // call to set_values inherited from mother class
  ppoly2->set_values (4,5); // same here

  cout << rect.area() << endl; // 20
  cout << trgl.area() << endl; // 10

  return 0;
}
```

Virtual members

- A member of a class that can be redefined in its derived classes is known as a virtual member

- Now the three classes (`CPolygon`, `CRectangle,` and `CTriangle`) have all the same members

- The member function `area( )` has been declared as virtual in the base class because it is later redefined in each derived class
  - If you remove the `virtual` keyword from the declaration of `area()` within CPolygon, the result will be 0 for the three polygons instead of 20, 10, and 0.
  - Expl: instead of calling the corresponding `area()` function for each object (`CRectangle::area()`, `CTriangle::area()`, and `CPolygon::area()`, respectively), the function `CPolygon::area()` will be called in all cases

- What the `virtual` keyword does?
  - Allows 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

- **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 on this example

Main program (virtual members)

```cpp
#include <iostream>
using namespace std;

class CPolygon {//mother class
  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 () {
//create objects and pointers for one rectangle, one triangle and one polygon
  CRectangle rect; CPolygon * ppoly1 = &rect;  ppoly1->set_values (4,5);
  CTriangle trgl;  CPolygon * ppoly2 = &trgl;  ppoly2->set_values (4,5);
  CPolygon poly;   CPolygon * ppoly3 = &poly;  ppoly3->set_values (4,5);

//same calls to the area function, but different results
  cout << ppoly1->area() << endl; // area() from CRectangle
  cout << ppoly2->area() << endl; // area() from CTriangle
  cout << ppoly3->area() << endl; // area() from CPolygon
  return 0;
}
```

Result

```
20
10
0
```

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 **must** leave that area() member function **without implementation at all**. This is done by appending =0 (equal to zero) to the function declaration :

```
// 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; //pure virtual function here
};
```

- 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.

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 it tries to instantiate an object
  - Nevertheless, the following pointers would be perfectly valid:

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

- This is so for as long as `CPolygon` includes a pure virtual function and therefore is an abstract base class

- However, pointers to this abstract base class can be used to point to objects of 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; }
    virtual int area (void) =0; // pure virtual: cannot instantiate Polygons
  };

class CRectangle: public CPolygon { //inherits from Cpolygon, area() will be here
  public:    int area (void)      { return (width * height); }
  };

class CTriangle: public CPolygon { //inherits from CTrianle, area() will be here
  public:    int area (void)     { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect; // look out here
  CPolygon * ppoly2 = &trgl; // look out here
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;  // call to CRectangle :: area - - - > 20
  cout << ppoly2->area() << endl;  // call to CTriangle :: area - - - > 10
  return 0;
}
```

Second example

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

class CPolygon { //abstract base class
  protected:    int width, height;
  public:    void set_values (int a, int b)      { width=a; height=b; }
             virtual int area (void) =0; // pure virtual function
             void printarea (void)               { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon { //as usual
  public:    int area (void)      { return (width * height); }
  };

class CTriangle: public CPolygon { //as usual
  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(); // what happens here?    // 20
  ppoly2->printarea(); // and here?             // 10

  return 0;
}
```

Third example

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

class CPolygon { // abstract base class
  protected:    int width, height;
  public:
    void set_values (int a, int b)      { width=a; height=b; }
    virtual int area (void) =0; //pure virtual function
    void printarea (void)               { cout << this->area() << endl; }
  };

class CRectangle: public CPolygon { // as usual
  public:    int area (void)      { return (width * height); }
  };

class CTriangle: public CPolygon { //as usual
  public:    int area (void)     { return (width * height / 2); }
  };

int main () {
  CPolygon * ppoly1 = new CRectangle; // call the constructor, gets a pointer
  CPolygon * ppoly2 = new CTriangle;  // but which constructor??
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1; // call the destructor
  delete ppoly2; // but which one?
  return 0;
}
```

- Look out:

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

- Notice that the `ppoly` pointers are declared being of type pointer to `CPolygon` but the objects dynamically allocated have been declared having the derived class type directly

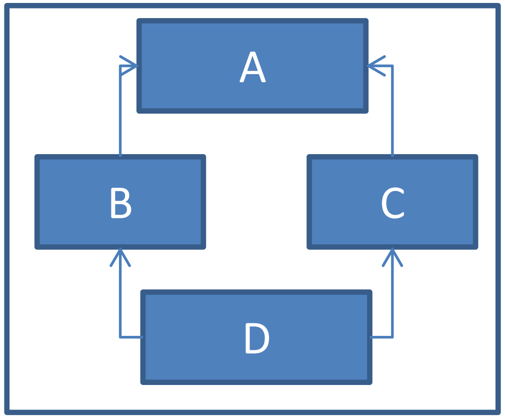
Virtual inheritance

- To solve ambiguity issues which arise when Multiple inheritances create an inheritance cycle, the simplest one being the diamond inheritance.

- Two strategies to get rid of ambiguities
  - Explicit call using the scope operator ::

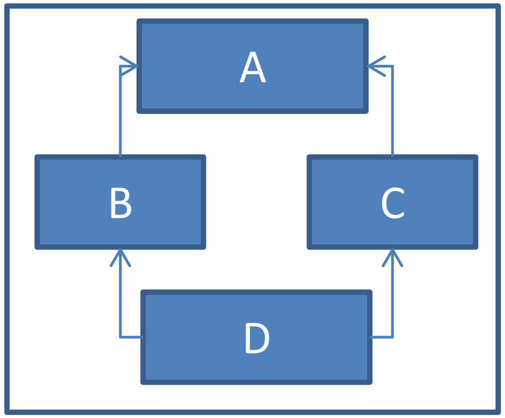
    → Exemple 1
  - Virtual inheritance

    → Exemple 2



Diamond inheritance

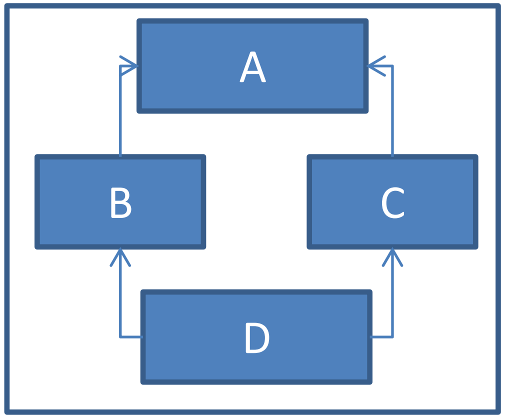
```
class A {
public :
int m_A;
void f ( ) {cout  << " A::f()" << endl; }
 A ( ) { cout << " A ()" << endl; }
~A ( ) { cout << "~A ()" << endl; }
};

class B : public A {
public :
int m_B;
 B ( ) { cout << " B ()" << endl; }
~B ( ) { cout << "~B ()" << endl; }
};

class C : public A{
public :
int m_C;
 C ( ) {cout << " C ()" << endl; }
~C ( ) {cout << "~C ()" << endl; }
};

class D : public B, public C {
public :
int m_D;
 D ( ) { cout << " D ()" << endl; }
~D ( ) { cout << "~D ()" << endl; }
};
```



Diamond inheritance

```
//create one instance of each class. Run to see the
    multiple calls to A() when creating object d.

A a; B b; C c; D d;

//call function f
a.f(); //OK
b.f(); //OK
c.f(); //OK

//next call creates confusion: B::f( ) or C::f( )
d.f(); // ERROR: because ambiguity

//solution : remove ambiguity through :: operator

cout <<"d.B::f() => "; d.B::f();
cout <<"d.C::f() => "; d.C::f();
```

```
class A {
public :
int m_A;
void f ( ) {cout  << " A::f()" << endl; }
 A ( ) { cout << " A ()" << endl; }
~A ( ) { cout << "~A ()" << endl; }
};

class B : virtual public A {
public :
int m_B;
 B ( ) { cout << " B ()" << endl; }
~B ( ) { cout << "~B ()" << endl; }
};

class C : virtual public A{
public :
int m_C;
 C ( ) {cout << " C ()" << endl; }
~C ( ) {cout << "~C ()" << endl; }
};

class D : public B, public C {
public :
int m_D;
 D ( ) { cout << " D ()" << endl; }
~D ( ) { cout << "~D ()" << endl; }
};
```



Diamond inheritance

```
//create one instance of each class
A a; B b; C c; D d;

//call function f
a.f(); //OK
b.f(); //OK
c.f(); //OK
d.f(); //Ok now, will use A :: f()
```

Outline

- Templates
  - Function templates
  - Class templates
  - Template specialization
  - Non-type parameters for templates
  - Templates and multiple-file projects

Function templates

- Special functions
  - can operate with **generic types.**
  - 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.

- Can be achieved using template parameters
  - 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

- Syntax:

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

- Differences between both prototypes:
  - 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:

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

- Creation of a template function with `myType` as its 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 value 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:
  - 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

Example

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

template <class T>        // template function
T GetMax (T a, T b) {     // return type? Arguments type?
   T result;              // what does this mean?
   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);  // call to the template function using int
  n = GetMax<long>(l,m); // same using long

  cout << k << endl;  // 6
  cout << n << endl;  //10

  return 0;
}
```

Explanations

- In the example we used the function template `GetMax()` twice:
  - The first time with arguments of type `int.` The second one with arguments of type `long.`
  - The compiler has instantiated and then called each time the appropriate version of the function.

- As you can see, 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:
  - The generic type T is used as a parameter for `GetMax()`
  - The compiler can find out automatically which data type has to be instantiated without having to explicitly specify it within angle brackets
  - So we could have written instead: `int i,j; GetMax (i,j);`

- 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

Discussion

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

```
    int i; long l;
    k = GetMax (i,l); //Error: i and l not of same type!
```

- 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);
```

- The problem here now becomes a conversion issue since the return type is of class T, which forces a conversion to be done (implicitely or explicitely) if the returned value is of class U.

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]; // private array of what?
  public:
    mypair (T first, T second) { values[0]=first; values[1]=second; }
};
```

- For instance, 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. See example next.

```
// class templates

template <class T>
class mypair {
    T a, b;
  public:
    mypair (T first, T second)      { a = first; b = second; }
    T getmax ( ); // declaration
};

// prefix required since not implemented in class declaration
template <class T>
T mypair<T> :: getmax () { return a > b ? a : b;}

int main ( ) {

  mypair <int> myobject (100, 75);
  cout << myobject.getmax(); //100
  return 0;
}
```

Comments

- 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
  - 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

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:

Source code

```
// 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 <> // look out here
class mycontainer <char> { // specialization
    char element; // no longer template
here!
  public:
    mycontainer (char arg) {element=arg;}
    char uppercase ()
    {
      if ((element>='a')&&(element<='z'))
      element+='A'-'a';
      return element;
    }
};
```

Result

```
int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() <<
endl;
  return 0;
}

-----------------------------------

8
J
```

Template specialization

- 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 **empty template < >** parameter list: 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> { ... };
```

- Notice that a specialized class must be fully redefined (if you do not reimplement a function, it will be considered as removed for the specialized type)

Non-type parameters for templates

```
template <class T, int N> // look our for N here!
class mysequence {
    T memblock [N]; // N used for the size
  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'; //100
  cout << myfloats.getmember(3) << '\n'; // 3.1416
  return 0;
}
```

Default parameters

- It is also possible to set default values or types for class template parameters similarly to what we did for default parameter for functions (using = operator in the declaration).

- 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<c har, 10> myseq;
```

Templates and multiple-file projects

- For the compiler, templates are not normal functions or classes. They are **compiled on demand**: the code of a template function is not compiled **until an instantiation** with specific template arguments **is required** : 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 files : in these cases, the declaration and definition 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.