Unit IV Contents

1 Principles or object oriented programming

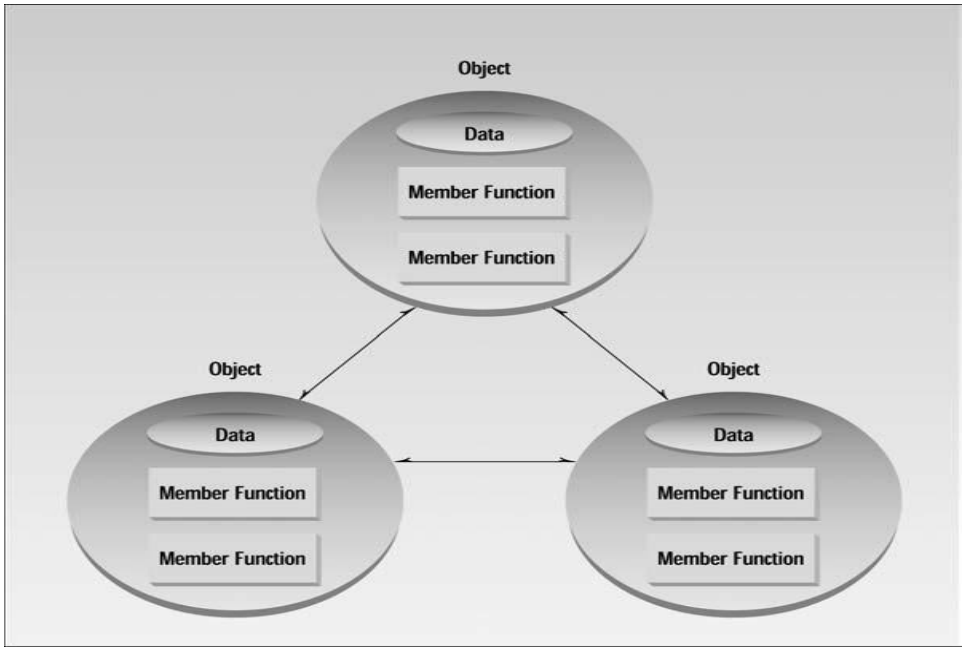
Why Do We Need Object-Oriented Programming?

Object-oriented programming was developed because limitations were discovered in earlier approaches to programming. To appreciate what OOP does, we need to understand what these limitations are and how they arose from traditional programming languages.

1.1 The Object-Oriented Approach

The fundamental idea behind object-oriented languages is to combine into a single unit both *data* and the *functions that operate on that data*. Such a unit is called an *object*.

An object's functions, called member functions in C++, typically provide the only way to access its data. If you want to read a data item in an object, you call a member function in the object. It will access the data and return the value to you. You can't access the data directly. The data is hidden, so it is safe from accidental alteration. Data and its functions are said to be encapsulated into a single entity. Data encapsulation and data hiding are key terms in the description of object-oriented languages. If you want to modify the data in an object, you know exactly what functions interact with it: the member functions in the object. No other functions can access the data. This simplifieswriting, debugging, and maintaining the program. A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions. The organization of a C++ program is shown in Figure 1.3: *The object-oriented paradigm.*



1.2 Characteristics of Object-Oriented Languages

- **Objects**

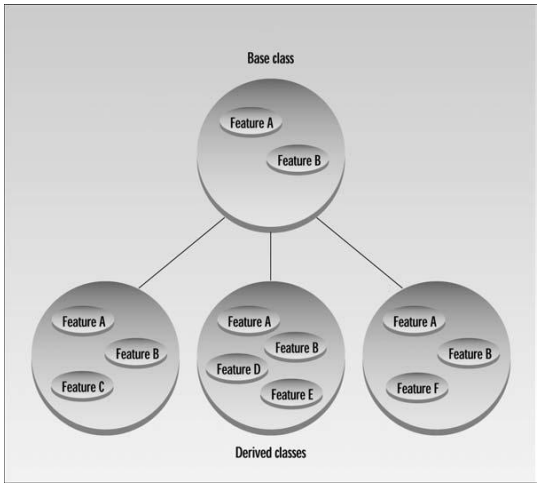
Examples are

• Physical objects Automobiles in a traffic-flow simulation Electrical components in a circuit-design program Countries in an economics model Aircraft in an air traffic control system	• Data-storage constructs Customized arrays Stacks Linked lists Binary trees	• Collections of data An inventory A personnel file A dictionary A table of the latitudes and longitudes of world cities	• Components in computer games Cars in an auto race Positions in a board game (chess, checkers) Animals in an ecological simulation Opponents and friends in adventure games
• Elements of the computer-user environment Windows Menus Graphics objects (lines, rectangles, circles) The mouse, keyboard, disk drives, printer	• Human entities Employees Students Customers Salespeople	• User-defined data types Time Angles Complex numbers Points on the plane	

- **Classes:**

A class is thus a description of a number of similar objects. It specifies what data and what functions will be included in objects of that class. Defining the class doesn't create any objects.

- **Inheritance**



Features A and B, which are part of the base class, are common to all the derived classes, but that each derived class also has features of its own.

- **Reusability:**

Once a class has been written, created, and debugged, it can be distributed to other programmers for use in their own programs. This is called *reusability*. It is similar to the way a library of functions in a procedural language can be incorporated into different programs.

- **Creating New Data Types**

One of the benefits of objects is that they give the programmer a convenient way to construct new data types, such as x and y coordinates, or latitude and longitude.

- **Polymorphism and Overloading**

When an existing operator, such as + or =, is given the capability to operate on a new data type, it is said to be overloaded. Overloading is a kind of polymorphism; it is also an important feature of OOP.

---

2 C++ Getting Started

2.1 Basic Program Construction

Let's look at a very simple C++ program. This program is called FIRST, so its source file is FIRST.CPP. It simply prints a sentence on the screen. Here it is:

```
#include <iostream>
using namespace std;
int main()
{
cout << "Every age has a language of its own\n";
return 0;
}
```

Functions

Functions are one of the fundamental building blocks of C++. The FIRST program consists almost entirely of a single function called main().

Function Name

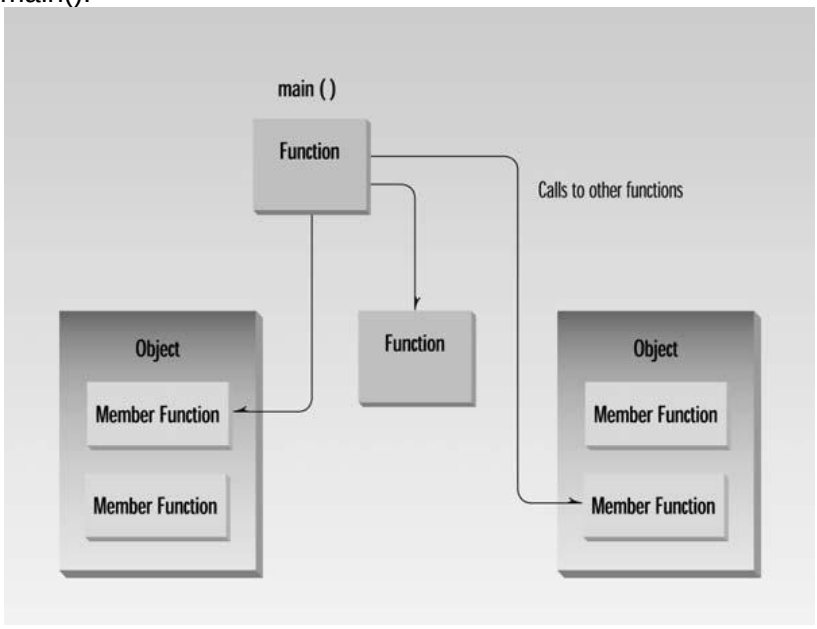
The parentheses following the word main are the distinguishing feature of a function. Without the parentheses the compiler would think that main refers to a variable or to some other program element.

Braces and the Function Body

The *body* of a function is surrounded by *braces* (sometimes called *curly brackets*). These braces play the same role as the BEGIN and END keywords in some other languages.

Always Start with main()

When you run a C++ program, the first statement executed will be at the beginning of a function called main().



Directives

The two lines that begin the FIRST program are *directives*. The first is a *preprocessor directive*, and the second is a using *directive*.

Preprocessor Directives

The first line of the FIRST program

#include <iostream>

A preprocessor directive, on the other hand, is an instruction to the *compiler*. A part of the compiler called the *preprocessor* deals with these directives before it begins the real compilation process. The preprocessor directive #include tells the compiler to insert another file into your source file. In effect, the #include directive is replaced by the contents of the file indicated.

The using Directive

A C++ program can be divided into different *namespaces*. A namespace is a part of the program in which certain names are recognized; outside of the namespace they're unknown. The directive ***using namespace std;*** says that all the program statements that follow are within the std namespace. Various program components such as cout are declared within this namespace. If we didn't use the using directive, we would need to add the std name to many program elements. For example, in the FIRST program we'd need to say std::cout << "Every age has a language of its own.";

Comments
The compiler ignores comments, so they do not add to the file size or execution time of the executable program.
Comment Syntax

```
// comments.cpp
// demonstrates comments
#include <iostream> //preprocessor directive
using namespace std; //"using" directive
/* ….multi
Line comment…. */
```

2.2 C++ Language Fundamentals: Tokens, Expressions, Classes

As we know that Software is a Program. And a Program is that which Contains set of instructions, and an Instruction contains Some Tokens. So Tokens are used for Writing the Programs the various Types of Tokens those are contained by the Programs.

2.2.1 Tokens

As in the English language, in a paragraph all the words, punctuation mark and the blank spaces are called ***Tokens***. Similarly in a C++ program all the C++ statements having *Keywords, Identifiers, Constants, Strings, Operators and the Special Symbols are called C++ Tokens.* C++ Tokens are the essential part of a C++ compiler and so are very useful in the C++ programming. A Token is an individual entity of a C++ program.

For example, some C++ Tokens used in a C++ program are:

Reserve words/ keywords: long, do if, else etc.

Identifiers: Pay, salary etc.

Constant: 470.6,16,49 etc.

Strings: "Dinesh", "2013-01" etc.

Operator: +, *, <, >=, &&,11, etc

Special symbols: 0, {}, #, @, %, etc.

2.2.2 Expressions

There are **three** types of expressions:

I. **Arithmetic expression**
II. **Relational expression**
III. **Logical expression**

What is a **C++ OPERATOR**?

"**C++ OPERATORS are signs use to perform certain task e.g addition**"

There are two kinds of operators:

a) Unary operator b) Binary operator

What is the difference between Unary and Binary operator?

Unary operators:

*"Requires **single operand item** to perform operation".*

Binary operators:

*"Required **more than one operand item** to perform operation".*

Types of operators :

- **Arithmetic operators**
- **Relational operators**
- **Logical operators**
- **Increment and decrement operators**
- **Assignment operator**
- **Bit-wise operator**

Arithmetic Expression and Arithmetic operator:

"**An expression in which arithmetic operators are used is called arithmetic expression**". For example an arithmetic expression is look just like that a+b=5

Explanation:

LIST OF ARITHMETIC OPERATORS AND THEIR FUNCTIONS

Operators	Function
+	Used for addition of two or more numbers
-	Used for subtraction of two or more numbers
*	Used for multiply two or more numbers
/	Used two divide numbers
%	this operator is used to get remainders

- These are used for all kind of numeric data.
- "%" is also called modulus operator it can be use only with integers.
- Unary operators has higher precedence as compared to binary operators.
- Multiplication (*) and Division(/) as higher priority than addition(+) and subtraction(-) where addition and division has equal priority.

Modes of Arithmetic Expressions

1.Mixed arithmetic
2.Real arithmetic
3.Integer arithmetic

Integer arithmetic mode
In this mode when arithmetic operation perform by using integer values it always result an integer value.
for example:
a=5 , b=5

a*b=25 , a/b=1 , a+b=10 , a-b=0
Real arithmetic mode
In this mode when a arithmetic operation is performed by using floating point numbers it always result an floating value.
a=10.0 , b=5.0

a*b=50.0 a/b=2.0 a+b=15.0 a-b=5.0
Mixed arithmetic mode
In this mode when an arithmetic operation performed on float and integer values it always result a float value.
For example:
a=10 , b=5.0
a*b=50.0,a/b=2.0,a+b=15.0,a-b=5.0
Relational operators and relational expressions
"A relational operator with constants and variables makes relational expression or An expressions in which relational operators are use is called relationalexpression."
Points about relational operators

- 1.Relational operators are use to compare values.
- 2.All the expressions evaluates from left to right.
- 3.There are six relational operators in C++ programming (>,< ,>=,<=,==,!=).
- 4.These operators evaluates results true or false.
- 5.False statement represent by 0 and True statement represent by 1.
- 6.These operators evaluate at statement level and has no preference.
-

Operator	Meaning
>	Greater than
<	Less than
>=	Greater than equal than
<=	Less than equal than
!=	not equal to
= =	Equal to (conditional operator)

- Logical Expression and Logical Operators

Logical operators

- 1.There are three logical operators And(&&),or(||) these two both are binary operator and not(!) is u nary operator.
- 2.More than one relation expression are combine by using logical operators.
- 3.The expression will evaluate from left to right if more than one relation expression are use.
- And operator (&&)

- It produces true result if all the expression or conditions are true.
- It produces false if any one expression is false.
- Below table shows evaluation method of and operator.

Exp-1	Exp-2	Result
1	1	1
1	0	0
0	1	0
0	0	0

- 1 represent True 0 represent false.
- Example to understanding of And (&&) operator.
  a=10,b=5

Exp-1	Exp-2	Result
(a>b) it evaluates 1	(b<a) it will evaluate 1	1
(a>b) it evaluates 1	(b>a) it will evaluate 0	0
(a<b) it evaluates 0	(b<a) it will evaluate 1	0
(a<b) it evaluates 0	(b>a) it will evaluate 0	0

- Or operator(||)

  It produces true if any expression is true.
- It produces false if all the conditions are false.
- Below table shows evaluation method of Or(||) operator.

Exp-1	Exp-2	Result
1	1	1
1	0	1
0	1	1
0	0	0

Example to understanding of And (||) operator.

a=10,b=5,

Exp-1	Exp-2	Result
(a>b) it evaluates 1	(b<a) it will evaluate 1	1
(a>b) it evaluates 1	(b>a) it will evaluate 0	1
(a<b) it evaluates 0	(b<a) it will evaluate 1	1
(a<b) it evaluates 0	(b>a) it will evaluate 0	0

-
- Not operator(!)

- 1.If expression provides true result it convert it into false.
- 2.If expression provides false result it convert it into true.
-

Evaluation method.

Result ! Result

1 0

0 1

Example to understanding of And (!) operator.

a=10,b=5

Result	! Result
(a>b) it will evaluates 1	0
(a<b) it will evaluate 0	1

2.2.3 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.

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:

```
1 class CRectangle {
2     int x, y;
3   public:
4     void set_values (int,int);
5     int area (void);
6   } 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 definition, 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 any 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`:

```
1  // example: one class, two
2  objects
3  #include <iostream>
4  using namespace std;
5
6  class CRectangle {
7      int x, y;
8    public:
9      void set_values (int,int);
10     int area () {return
11 (x*y);}
12 };
13
14 void CRectangle::set_values
15 (int a, int b) {
```

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

```
  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;
}
```

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`.

2.3 Functions, Constructors, Destructors

2.3.1 Functions

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming can offer to us in C++.

A function is a group of statements that is executed when it is called from some point of the program. The following is its format:

type name (parameter1, parameter2, ...) { statements }

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

Here you have the first function example:

```
1   // function example
2   #include <iostream>
3   using namespace std;
4
5   int addition (int a, int b)
6   {
7     int r;
8     r=a+b;
9     return (r);
10  }
11
12  int main ()
13  {
14    int z;
15    z = addition (5,3);
16    cout << "The result is " << z;
17    return 0;
18  }
```

```
The result is 8
```

In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution by the main function. So we will begin there.

We can see how the main function begins by declaring the variable z of type int. Right after that, we see a call to a function called addition. Paying attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself some code lines above:

```
int addition (int a, int b)
                 ↑      ↑
z = addition (   5   ,    3   );
```

The parameters and arguments have a clear correspondence. Within the main function we called to additionpassing two values: 5 and 3, that correspond to the int a and int b parameters declared for function addition.

At the point at which the function is called from within main, the control is lost by main and passed to functionaddition. The value of both arguments passed in the call (5 and 3) are copied to the local variables int a and int b within the function.

Function addition declares another local variable (int r), and by means of the expression r=a+b, it assigns to rthe result of a plus b. Because the actual parameters passed for a and b are 5 and 3 respectively, the result is 8.

The following line of code:

```
return (r);
```

finalizes function addition, and returns the control back to the function that called it in the first place (in this case,main). At this moment the program follows its regular course from the same point at which it was interrupted by the call to addition. But additionally, because the return statement in function addition specified a value: the content of variable r (return (r);), which at that moment had a value of 8. This value becomes the value of evaluating the function call.

```
int addition (int a, int b)
 8
z = addition (  5  ,  3  );
```

So being the value returned by a function the value given to the function call itself when it is evaluated, the variable z will be set to the value returned by addition (5, 3), that is 8. To explain it another way, you can imagine that the call to a function (addition (5,3)) is literally replaced by the value it returns (8).

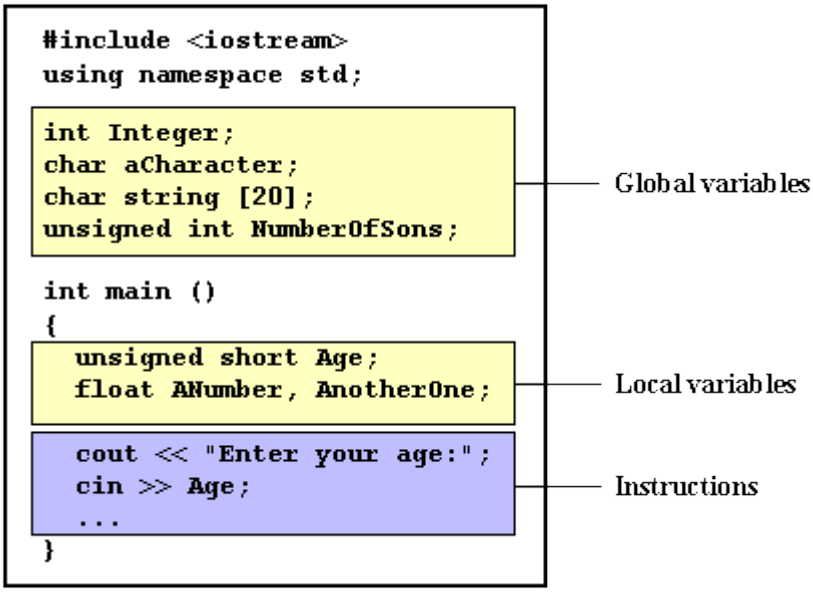
The following line of code in main is:

```
cout << "The result is " << z;
```

That, as you may already expect, produces the printing of the result on the screen.

Scope of variables

The scope of variables declared within a function or any other inner block is only their own function or their own block and cannot be used outside of them. For example, in the previous example it would have been impossible to use the variables a, b or r directly in function main since they were variables local to function addition. Also, it would have been impossible to use the variable z directly within function addition, since this was a variable local to the function main.



Therefore, the scope of local variables is limited to the same block level in which they are declared. Nevertheless, we also have the possibility to declare global variables; These are visible from any point of the code, inside and

outside all functions. In order to declare global variables you simply have to declare the variable outside any function or block; that means, directly in the body of the program.

And here is another example about functions:

```
1   // function example
2   #include <iostream>
3   using namespace std;
4
5   int subtraction (int a, int b)
6   {
7     int r;
8     r=a-b;
9     return (r);
10  }
11
12  int main ()
13  {
14    int x=5, y=3, z;
15    z = subtraction (7,2);
16    cout << "The first result is " << z << '\n';
17    cout << "The second result is " << subtraction (7,2) << '\n';
18    cout << "The third result is " << subtraction (x,y) << '\n';
19    z= 4 + subtraction (x,y);
20    cout << "The fourth result is " << z << '\n';
21    return 0;
22  }
```

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

In this case we have created a function called subtraction. The only thing that this function does is to subtract both passed parameters and to return the result.

Nevertheless, if we examine function main we will see that we have made several calls to function subtraction. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to fully understand these examples you must consider once again that a call to a function could be replaced by the value that the function call itself is going to return. For example, the first case (that you should already know because it is the same pattern that we have used in previous examples):

```
1 z = subtraction (7,2);
2 cout << "The first result is " << z;
```

If we replace the function call by the value it returns (i.e., 5), we would have:

```
1 z = 5;
2 cout << "The first result is " << z;
```

As well as

```
cout << "The second result is " << subtraction (7,2);
```

has the same result as the previous call, but in this case we made the call to subtraction directly as an insertion parameter for cout. Simply consider that the result is the same as if we had written:

```
cout << "The second result is " << 5;
```

since 5 is the value returned by subtraction (7,2).

In the case of:

```
cout << "The third result is " << subtraction (x,y);
```

The only new thing that we introduced is that the parameters of subtraction are variables instead of constants. That is perfectly valid. In this case the values passed to function subtraction are the values of x and y, that are 5and 3 respectively, giving 2 as result.

The fourth case is more of the same. Simply note that instead of:

```
z = 4 + subtraction (x,y);
```

we could have written:

```
z = subtraction (x,y) + 4;
```

with exactly the same result. I have switched places so you can see that the semicolon sign (;) goes at the end of the whole statement. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its returned value:

```
1 z = 4 + 2;
2 z = 2 + 4;
```

Functions with no type. The use of void.

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

you will see that the declaration begins with a type, that is the type of the function itself (i.e., the type of the datum that will be returned by the function with the return statement). But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value. In this case we should use the void type specifier for the function. This is a special specifier that indicates

absence of type.

```
1   // void function example
2   #include <iostream>
3   using namespace std;
4
5   void printmessage ()
6   {
7     cout << "I'm a function!";
8   }
9
10  int main ()
11  {
12    printmessage ();
13    return 0;
14  }
```

```
I'm a function!
```

void can also be used in the function's parameter list to explicitly specify that we want the function to take no actual parameters when it is called. For example, function printmessage could have been declared as:

```
1 void printmessage (void)
2 {
3   cout << "I'm a function!";
4 }
```

Although it is optional to specify void in the parameter list. In C++, a parameter list can simply be left blank if we want a function with no parameters.

What you must always remember is that the format for calling a function includes specifying its name and enclosing its parameters between parentheses. The non-existence of parameters does not exempt us from the obligation to write the parentheses. For that reason the call to printmessage is:

```
printmessage ();
```

The parentheses clearly indicate that this is a call to a function and not the name of a variable or some other C++ statement. The following call would have been incorrect:

```
printmessage;
```

Arguments passed by value and by reference.

Until now, in all the functions we have seen, the arguments passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were copies of their values but never the variables themselves. For example, suppose that we called our first function additionusing the following code:

```
1 int x=5, y=3, z;
2 z = addition ( x , y );
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

```
int addition (int a, int b)
                 ↑      ↑
z = addition (   5   ,   3   );
```

This way, when the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and youtside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we can use arguments passed by reference, as in the function duplicate of the following example:

```
1   // passing parameters by reference
2   #include <iostream>
3   using namespace std;
4
5   void duplicate (int& a, int& b, int& c)
6   {
7     a*=2;
8     b*=2;
9     c*=2;
10  }
11
12  int main ()
13  {
14    int x=1, y=3, z=7;
15    duplicate (x, y, z);
16    cout << "x=" << x << ", y=" << y << ", z=" << z;
17    return 0;
18  }
```

```
x=2, y=6, z=14
```

The first thing that should call your attention is that in the declaration of duplicate the type of each parameter was followed by an ampersand sign (&). This ampersand is what specifies that their corresponding arguments are to be passed *by reference* instead of *by value*.

When a variable is passed by reference we are not passing a copy of its value, but we are somehow passing the variable itself to the function and any modification that we do to the local variables will have an effect in their counterpart variables passed as arguments in the call to the function.

```
void duplicate (int& a,int& b,int& c)
                  ↕x    ↕y    ↕z
     duplicate (  x  ,  y  ,  z  );
```

To explain it in another way, we associate a, b and c with the arguments passed on the function call (x, y and z) and any change that we do on a within the function will affect the value of x outside it. Any change that we do on bwill affect y, and the same with c and z.

That is why our program's output, that shows the values stored in x, y and z after the call to duplicate, shows the values of all the three variables of main doubled.

If when declaring the following function:

```
void duplicate (int& a, int& b, int& c)
```

we had declared it this way:

```
void duplicate (int a, int b, int c)
```

i.e., without the ampersand signs (&), we would have not passed the variables by reference, but a copy of their values instead, and therefore, the output on screen of our program would have been the values of x, y and zwithout having been modified.

Passing by reference is also an effective way to allow a function to return more than one value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
1  // more than one returning value
2  #include <iostream>
3  using namespace std;
4
5  void prevnext (int x, int& prev, int& next)
6  {
7    prev = x-1;
8    next = x+1;
9  }
10
11 int main ()
12 {
13   int x=100, y, z;
14   prevnext (x, y, z);
15   cout << "Previous=" << y << ", Next=" << z;
16   return 0;
17 }
```

```
Previous=99, Next=101
```

Default values in parameters.
When declaring a function we can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
1  // default values in functions
2  #include <iostream>
3  using namespace std;
```

```
6
5
```

```
4
5   int divide (int a, int b=2)
6   {
7     int r;
8     r=a/b;
9     return (r);
10  }
11
12  int main ()
13  {
14    cout << divide (12);
15    cout << endl;
16    cout << divide (20,4);
17    return 0;
18  }
```

As we can see in the body of the program there are two calls to function divide. In the first one:

```
divide (12)
```

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

Overloaded functions.

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. For example:

```
1   // overloaded function
2   #include <iostream>
3   using namespace std;
4
5   int operate (int a, int b)
6   {
7     return (a*b);
8   }
9
10  float operate (float a, float b)
```

```
10
2.5
```

```
11  {
12    return (a/b);
13  }
14
15  int main ()
16  {
17    int x=5,y=2;
18    float n=5.0,m=2.0;
19    cout << operate (x,y);
20    cout << "\n";
21    cout << operate (n,m);
22    cout << "\n";
23    return 0;
24  }
```

In this case we have defined two functions with the same name, operate, but one of them accepts two parameters of type int and the other one accepts them of type float. The compiler knows which one to call in each case by examining the types passed as arguments when the function is called. If it is called with two ints as its arguments it calls to the function that has two int parameters in its prototype and if it is called with two floats it will call to the one which has two float parameters in its prototype.

In the first call to operate the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. This one has a different behavior: it divides one parameter by the other. So the behavior of a call to operate depends on the type of the arguments passed because the function has been *overloaded*.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

inline functions.

The inline specifier indicates the compiler that inline substitution is preferred to the usual function call mechanism for a specific function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

The format for its declaration is:

inline type name (arguments ...) { instructions ... }

and the call is just like the call to any other function. You do not have to include the inline keyword when calling the function, only in its declaration.

Most compilers already optimize code to generate inline functions when it is more convenient. This specifier only indicates the compiler that inline is preferred for this function.

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for many tasks, like sorting or

calculate the factorial of numbers. For example, to obtain the factorial of a number (n!) the mathematical formula would be:

n! = n * (n-1) * (n-2) * (n-3) ... * 1

more concretely, 5! (factorial of 5) would be:

5! = 5 * 4 * 3 * 2 * 1 = 120

and a recursive function to calculate this in C++ could be:

```
1   // factorial calculator
2   #include <iostream>
3   using namespace std;
4
5   long factorial (long a)
6   {
7     if (a > 1)
8       return (a * factorial (a-1));
9     else
10      return (1);
11  }
12
13  int main ()
14  {
15    long number;
16    cout << "Please type a number: ";
17    cin >> number;
18    cout << number << "! = " << factorial (number);
19    return 0;
20  }
```

```
Please type a number: 9
9! = 362880
```

Notice how in function factorial we included a call to itself, but only if the argument passed was greater than 1, since otherwise the function would perform an infinite recursive loop in which once it arrived to 0 it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime).

This function has a limitation because of the data type we used in its design (long) for more simplicity. The results given will not be valid for values much greater than 10! or 15!, depending on the system you compile it.

Declaring functions.

Until now, we have defined all of the functions before the first appearance of calls to them in the source code. These calls were generally in function main which we have always left at the end of the source code. If you try to repeat some of the examples of functions described so far, but placing the function main before any of the other functions that were called from within it, you will most likely obtain compiling errors. The reason is that to be able to call a function it must have been declared in some earlier point of the code, like we have done in all our examples.

But there is an alternative way to avoid writing the whole code of a function before it can be used in main or in some other function. This can be achieved by declaring just a prototype of the function before it is used, instead of the

entire definition. This declaration is shorter than the entire definition, but significant enough for the compiler to determine its return type and the types of its parameters.

Its form is:

<return type> <function name> (argument_type1, argument_type2, ...);

It is identical to a function definition, except that it does not include the body of the function itself (i.e., the function statements that in normal definitions are enclosed in braces { }) and instead of that we end the prototype declaration with a mandatory semicolon (;).

The parameter enumeration does not need to include the identifiers, but only the type specifiers. The inclusion of a name for each parameter as in the function definition is optional in the prototype declaration. For example, we can declare a function called protofunction with two int parameters with any of the following declarations:

```
1 int protofunction (int first, int second);
2 int protofunction (int, int);
```

Anyway, including a name for each variable makes the prototype more legible.

```
1   // declaring functions prototypes
2   #include <iostream>
3   using namespace std;
4
5   void odd (int a);
6   void even (int a);
7
8   int main ()
9   {
10    int i;
11    do {
12      cout << "Type a number (0 to exit): ";
13      cin >> i;
14      odd (i);
15    } while (i!=0);
16    return 0;
17  }
18
19  void odd (int a)
20  {
21    if ((a%2)!=0) cout << "Number is odd.\n";
22    else even (a);
23  }
24
25  void even (int a)
26  {
27    if ((a%2)==0) cout << "Number is even.\n";
28    else odd (a);
29  }
```

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

This example is indeed not an example of efficiency. I am sure that at this point you can already make a program with the same result, but using only half of the code lines that have been used in this example. Anyway this example illustrates how prototyping works. Moreover, in this concrete example the prototyping of at least one of the two functions is necessary in order to compile the code without errors.

The first things that we see are the declaration of functions odd and even:

```
1 void odd (int a);
2 void even (int a);
```

This allows these functions to be used before they are defined, for example, in main, which now is located where some people find it to be a more logical place for the start of a program: the beginning of the source code.

Anyway, the reason why this program needs at least one of the functions to be declared before it is defined is because in odd there is a call to even and in even there is a call to odd. If none of the two functions had been previously declared, a compilation error would happen, since either odd would not be visible from even (because it has still not been declared), or even would not be visible from odd (for the same reason).

Having the prototype of all functions together in the same place within the source code is found practical by some programmers, and this can be easily achieved by declaring all functions prototypes at the beginning of a program.

2.4 The main() function

2.5 When a program begins running, the system calls the function `main`, which marks the entry point of the program. By default, `main` has the storage class `extern`. Every program must have one function named `main`

The function `main` can be defined with or without parameters, using any of the following forms:

```
int main (void)
int main ( )
int main(int argc, char *argv[])
int main (int argc, char ** argv)
```

The first parameter, `argc` (argument count) is an integer that indicates how many arguments were entered on the command line when the program was started. The second parameter, `argv`(argument vector), is an array of pointers to arrays of character objects.

2.6 Constructors

When an object of a class is created, C++ calls the *constructor* for that class. If no constructor is defined, C++ invokes a *default constructor*, which allocates memory for the object, but *doesn't initialize it*.

Why you should define a constructor

Uninitialized member fields have *garbage* in them. This creates the possibility of a serious bug (eg, an uninitialized pointer, illegal values, inconsistent values, ...).

2.6.1 Declaring a constructor

A constructor is similar to a function, but with the following differences.

- No return type.
- No return statement.

Example [extracts from three different files].

```
//=== point/point.h ================================================
#ifndef POINT_H
#define POINT_H
class Point {
   public:
      Point();  // parameterless default constructor
      Point(int new_x, int new_y); //  constructor with parameters
      int getX();
      int getY();
   private:
      int x;
      int y;
};
#endif
```

Here is part of the implementation file.

```
//=== point/point.cpp ==============================================
. . .
Point::Point() {  // default constructor
   x = 0;
   y = 0;
}

Point::Point(int new_x, int new_y) {  // constructor
   x = new_x;
   y = new_y;
}
. . .
```

And here is part of a file that uses the Point class.

```
//=== point/main.cpp ===============================================
  . . .
  Point p;            // calls our default constructor
  Point q(10,20);     // calls constructor with parameters
  Point* r = new Point();  // calls default constructor
  Point s = p;        // our default constructor not called.
  . . .
```

Copy Constructors

2.7 When copies of objects are made

A *copy constructor* is called whenever a new variable is created from an object. This happens in the following cases (but not in assignment).

- A variable is declared which is *initialized from another object*, eg,

```
• Person q("Mickey"); // constructor is used to build q.
• Person r(p);        // copy constructor is used to build r.
• Person p = q;       // copy constructor is used to initialize in
  declaration.
```

```
p = q;                  // Assignment operator, no constructor or copy
constructor.
```

- A value parameter is initialized from its corresponding argument.

```
f(p);                   // copy constructor initializes formal value
parameter.
```

- An object is returned by a function.

C++ calls a *copy constructor* to make a copy of an object in each of the above cases. If there is no copy constructor defined for the class, C++ uses the default copy constructor which copies each field, ie, makes a*shallow copy*.

2.8 Copy constructor syntax

The copy constructor takes a reference to a `const` parameter. It is `const` to guarantee that the copy constructor doesn't change it, and it is a reference because a value parameter would require making a copy, which would invoke the copy constructor, which would make a copy of its parameter, which would invoke the copy constructor, which ...

Here is an example of a copy constructor for the Point class, which doesn't really need one because the default copy constructor's action of copying fields would work fine, but it shows how it works.

```
//=== file Point.h ============================================
class Point {
    public:
        . . .
        Point(const Point& p);   // copy constructor
        . . .
//=== file Point.cpp ==========================================
. . .
Point::Point(const Point& p) {
    x = p.x;
    y = p.y;
}
    . . .
//=== file my_program.cpp =====================================
. . .
Point p;            // calls default constructor
Point s = p;        // calls copy constructor.
p = s;              // assignment, not copy constructor.
```

2.8.1 Destructors

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.

A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared `const`, `volatile`, `const volatile` or `static`. A destructor can be declared `virtual` or pure `virtual`.

If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.

The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class `A` has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for`A`:

```
  A::~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class `A` before defining the implicitly declared destructor of `A`

A destructor of a class `A` is *trivial* if all the following are true:

- It is implicitly defined
- All the direct base classes of `A` have trivial destructors
- The classes of all the nonstatic data members of `A` have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.

A union member cannot be of a class type that has a nontrivial destructor.

Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class `A` or a member of `A` has a destructor, and a class derived from `A` does not declare a destructor, a default destructor is generated.

The default destructor calls the destructors of the base class and members of the derived class.

The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.

Destructors are implicitly called when an automatic object (a local object that has been declared `auto` or `register`, or not declared as `static` or`extern`) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the `delete` operator for objects created with the `new` operator.

For example:

```
#include <string>

class Y {
private:
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
  // Destructor
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // ...

  // Destructor ~Y is called before
  // control returns from main()
}
```

You can use a destructor explicitly to destroy objects, although this practice is not recommended. However to destroy an object created with the placement `new` operator, you can explicitly call the object's destructor. The following example demonstrates this:

```
#include <new>
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A::A()" << endl; }
    ~A() { cout << "A::~A()" << endl; }
};
int main () {
  char* p = new char[sizeof(A)];
  A* ap = new (p) A;
```

```
  ap->A::~A();
  delete [] p;
}
```

The statement `A* ap = new (p) A` dynamically creates a new object of type `A` not in the free store but in the memory allocated by `p`. The statement `delete [] p` will delete the storage allocated by `p`, but the run time will still believe that the object pointed to by `ap` still exists until you explicitly call the destructor of `A` (with the statement `ap->A::~A()`).

2.8.2 Functions overloading

You overload a function name `f` by declaring more than one function with the name `f` in the same scope. The declarations of `f` must differ from each other by the types and/or the number of arguments in the argument list. When you call an overloaded function named `f`, the correct function is selected by comparing the argument list of the function call with the parameter list of each of the overloaded candidate functions with the name `f`. A *candidate function* is a function that can be called based on the context of the call of the overloaded function name.

Consider a function `print`, which displays an `int`. As shown in the following example, you can overload the function `print` to display other types, for example, `double` and `char*`. You can have three functions with the same name, each performing a similar operation on a different data type:

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}

void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
}
```

The following is the output of the above example:

```
 Here is int 10
 Here is float 10.1
 Here is char* ten
```

2.8.3 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, thanks to the 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);
```

```
4,3
```

```
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

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, I have to warn you that 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.

2.9 Pointers

We have already seen how variables are seen as memory cells that can be accessed using their identifiers. This way we did not have to care about the physical location of our data within memory, we simply used its identifier whenever we wanted to refer to our variable.

The memory of your computer can be imagined as a succession of memory cells, each one of the minimal size that computers manage (one byte). These single-byte memory cells are numbered in a consecutive way, so as, within any block of memory, every cell has the same number as the previous one plus one.

This way, each cell can be easily located in the memory because it has a unique address and all the memory cells follow a successive pattern. For example, if we are looking for cell 1776 we know that it is going to be right between cells 1775 and 1777, exactly one thousand cells after 776 and exactly one thousand cells before cell 2776.

2.9.1 Reference operator (&)

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.

The address that locates a variable within memory is what we call a *reference* to that variable. This reference to a variable can be obtained by preceding the identifier of a variable with an ampersand sign (`&`), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```

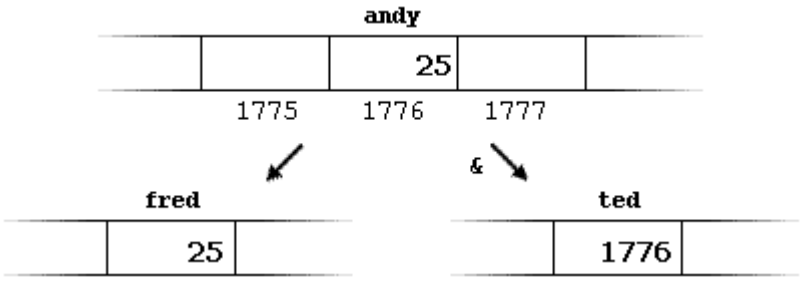
This would assign to `ted` the address of variable `andy`, since when preceding the name of the variable `andy` with the reference operator (`&`) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on we are going to assume that `andy` is placed during runtime in the memory address `1776`. This number (`1776`) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```
andy = 25;
fred = andy;
ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:



First, we have assigned the value 25 to `andy` (a variable whose address in memory we have assumed to be 1776).

The second statement copied to `fred` the content of variable `andy` (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to `ted` not the value contained in `andy` but a reference to it (i.e., its address, which we have assumed to be `1776`). The reason is that in this third assignment operation we have preceded the identifier `andy` with the reference operator (`&`), so we were no longer referring to the value of andy but to its reference (its address in memory).

The variable that stores the reference to another variable (like `ted` in the previous example) is what we call a*pointer*. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

2.9.2Dereference operator (*)

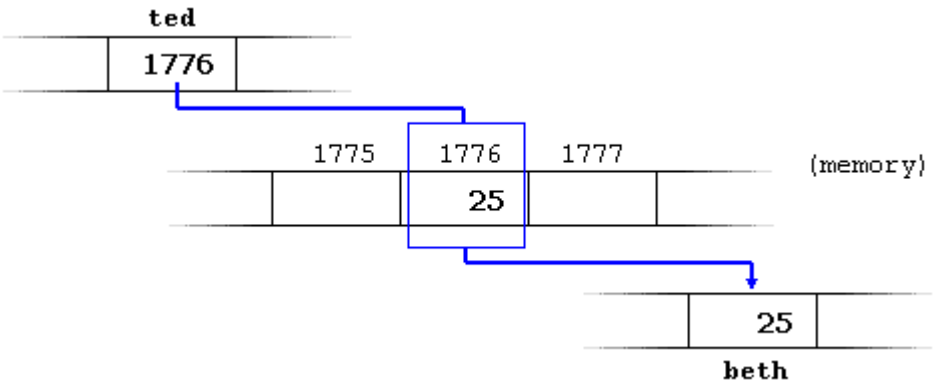
We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "`beth` equal to value pointed by `ted`") `beth` would take the value `25`, since `ted` is `1776`, and the value pointed by 1776 is 25.



You must clearly differentiate that the expression `ted` refers to the value `1776`, while `*ted` (with an asterisk `*`preceding the identifier) refers to the value stored at address `1776`, which in this case is `25`. Notice the difference of including or not including the dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

```
beth = ted;    // beth equal to ted ( 1776 )
beth = *ted;   // beth equal to value pointed by ted ( 25 )
```

Notice the difference between the reference and dereference operators:

- & is the reference operator and can be read as "address of"
- * is the dereference operator and can be read as "value pointed by"

Thus, they have complementary (or opposite) meanings. A variable referenced with `&` can be dereferenced with `*`.

Earlier we performed the following two assignment operations:

```
andy = 25;
ted = &andy;
```

Right after these two statements, all of the following expressions would give true as result:

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

The first expression is quite clear considering that the assignment operation performed on `andy` was `andy=25`. The second one uses the reference operator (`&`), which returns the address of variable `andy`, which we assumed it to have a value of `1776`. The third one is somewhat obvious since the second expression was true and the assignment operation performed on `ted` was `ted=&andy`. The fourth expression uses the dereference operator (`*`) that, as we have just seen, can be read as "value pointed by", and the value pointed by `ted` is indeed `25`.

So, after all that, you may also infer that for as long as the address pointed by `ted` remains unchanged the following expression will also be true:

```
*ted == andy
```

2.9.3 Declaring variables of pointer types

Due to the ability of a pointer to directly refer to the value that it points to, it becomes necessary to specify in its declaration which data type a pointer is going to point to. It is not the same thing to point to a `char` as to point to an `int` or a `float`.

The declaration of pointers follows this format:

`type * name;`

where `type` is the data type of the value that the pointer is intended to point to. This type is not the type of the pointer itself! but the type of the data the pointer points to. For example:

```
int * number;
char * character;
float * greatnumber;
```

These are three declarations of pointers. Each one is intended to point to a different data type, but in fact all of them are pointers and all of them will occupy the same amount of space in memory (the size in memory of a

pointer depends on the platform where the code is going to run). Nevertheless, the data to which they point to do not occupy the same amount of space nor are of the same type: the first one points to an `int`, the second one to a`char` and the last one to a `float`. Therefore, although these three example variables are all of them pointers which occupy the same size in memory, they are said to have different types: `int*`, `char*` and `float*` respectively, depending on the type they point to.

I want to emphasize that the asterisk sign (`*`) that we use when declaring a pointer only means that it is a pointer (it is part of its type compound specifier), and should not be confused with the dereference operator that we have seen a bit earlier, but which is also written with an asterisk (`*`). They are simply two different things represented with the same sign.

Now have a look at this code:

```
// my first pointer
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue, secondvalue;
  int * mypointer;

  mypointer = &firstvalue;
  *mypointer = 10;
  mypointer = &secondvalue;
  *mypointer = 20;
  cout << "firstvalue is " << firstvalue <<
endl;
  cout << "secondvalue is " << secondvalue
<< endl;
  return 0;
}
```

```
firstvalue is 10
secondvalue is 20
```

Notice that even though we have never directly set a value to either `firstvalue` or `secondvalue`, both end up with a value set indirectly through the use of `mypointer`. This is the procedure:

First, we have assigned as value of `mypointer` a reference to `firstvalue` using the reference operator (`&`). And then we have assigned the value 10 to the memory location pointed by `mypointer`, that because at this moment is pointing to the memory location of `firstvalue`, this in fact modifies the value of `firstvalue`.

In order to demonstrate that a pointer may take several different values during the same program I have repeated the process with `secondvalue` and that same pointer, `mypointer`.

Here is an example a little bit more elaborated:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int firstvalue = 5, secondvalue = 15;
  int * p1, * p2;

  p1 = &firstvalue;  // p1 = address of firstvalue
  p2 = &secondvalue; // p2 = address of secondvalue
  *p1 = 10;          // value pointed by p1 = 10
  *p2 = *p1;         // value pointed by p2 = value
pointed by p1
  p1 = p2;           // p1 = p2 (value of pointer is
```

```
firstvalue is 10
secondvalue is 20
```

```
16 copied)
17   *p1 = 20;          // value pointed by p1 = 20
18
19   cout << "firstvalue is " << firstvalue << endl;
20   cout << "secondvalue is " << secondvalue << endl;
     return 0;
}
```

I have included as a comment on each line how the code can be read: ampersand (`&`) as "address of" and asterisk (`*`) as "value pointed by".

Notice that there are expressions with pointers `p1` and `p2`, both with and without dereference operator (`*`). The meaning of an expression using the dereference operator (`*`) is very different from one that does not: When this operator precedes the pointer name, the expression refers to the value being pointed, while when a pointer name appears without this operator, it refers to the value of the pointer itself (i.e. the address of what the pointer is pointing to).

Another thing that may call your attention is the line:

```
int * p1, * p2;
```

This declares the two pointers used in the previous example. But notice that there is an asterisk (*) for each pointer, in order for both to have type `int*` (pointer to `int`).

Otherwise, the type for the second variable declared in that line would have been `int` (and not `int*`) because of precedence relationships. If we had written:

```
int * p1, p2;
```

`p1` would indeed have `int*` type, but `p2` would have type `int` (spaces do not matter at all for this purpose). This is due to operator precedence rules. But anyway, simply remembering that you have to put one asterisk per pointer is enough for most pointer users.

2.9.4 Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations:

```
1 int numbers [20];
2 int * p;
```

The following assignment operation would be valid:

```
p = numbers;
```

After that, `p` and `numbers` would be equivalent and would have the same properties. The only difference is that we could change the value of pointer `p` by another one, whereas `numbers` will always point to the first of the 20 elements of type `int` with which it was defined. Therefore, unlike `p`, which is an ordinary pointer, `numbers` is an array, and an array can be considered a *constant pointer*. Therefore, the following allocation would not be valid:

```
numbers = p;
```

Because `numbers` is an array, so it operates as a constant pointer, and we cannot assign values to constants.

Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

```
// more pointers
#include <iostream>
using namespace std;

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

```
10, 20, 30, 40, 50,
```

In the chapter about arrays we used brackets (`[]`) several times in order to specify the index of an element of the array to which we wanted to refer. Well, these bracket sign operators `[]` are also a **dereference** operator known as *offset operator*. They dereference the variable they follow just as `*` does, but they also add the number between brackets to the address being dereferenced. For example:

```
a[5] = 0;       // a [offset of 5] = 0
*(a+5) = 0;     // pointed by (a+5) = 0
```

These two expressions are equivalent and valid both if `a` is a pointer or if `a` is an array.

2.9.5 Pointer initialization

When declaring pointers we may want to explicitly specify which variable we want them to point to:

```
int number;
int *tommy = &number;
```

The behavior of this code is equivalent to:

```
int number;
int *tommy;
tommy = &number;
```

When a pointer initialization takes place we are always assigning the reference value to where the pointer points (`tommy`), never the value being pointed (`*tommy`). You must consider that at the moment of declaring a pointer, the asterisk (`*`) indicates only that it is a pointer, it is not the dereference operator (although both use the same sign: *). Remember, they are two different functions of one sign. Thus, we must take care not to confuse the previous code with:

```
int number;
```

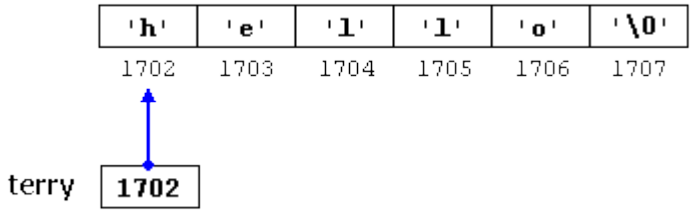
```
int *tommy;
*tommy = &number;
```

that is incorrect, and anyway would not have much sense in this case if you think about it.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment the pointer is declared:

```
const char * terry = "hello";
```

In this case, memory space is reserved to contain `"hello"` and then a pointer to the first character of this memory block is assigned to `terry`. If we imagine that `"hello"` is stored at the memory locations that start at addresses 1702, we can represent the previous declaration as:



It is important to indicate that `terry` contains the value 1702, and not `'h'` nor `"hello"`, although 1702 indeed is the address of both of these.

The pointer `terry` points to a sequence of characters and can be read as if it was an array (remember that an array is just like a constant pointer). For example, we can access the fifth element of the array with any of these two expression:

```
*(terry+4)
terry[4]
```

Both expressions have a value of `'o'` (the fifth element of the array).

2.9.6 Pointer arithmetics

To conduct arithmetical operations on pointers is a little different than to conduct them on regular integer data types. To begin with, only addition and subtraction operations are allowed to be conducted with them, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point.

When we saw the different fundamental data types, we saw that some occupy more or less space than others in the memory. For example, let's assume that in a given compiler for a specific machine, `char` takes 1 byte, `short`takes 2 bytes and `long` takes 4.

Suppose that we define three pointers in this compiler:

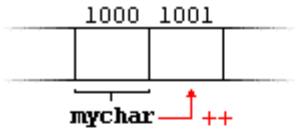
```
char *mychar;
short *myshort;
long *mylong;
```

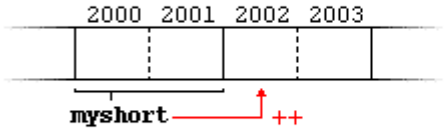
and that we know that they point to memory locations `1000`, `2000` and `3000` respectively.

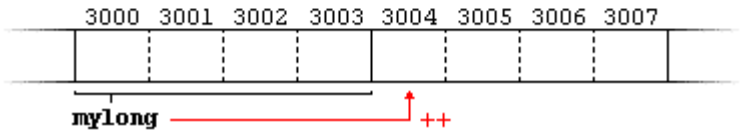
So if we write:

```
1 mychar++;
2 myshort++;
3 mylong++;
```

mychar, as you may expect, would contain the value 1001. But not so obviously, myshort would contain the value2002, and mylong would contain 3004, even though they have each been increased only once. The reason is that when adding one to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in bytes of the type pointed is added to the pointer.







This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
1 mychar = mychar + 1;
2 myshort = myshort + 1;
3 mylong = mylong + 1;
```

Both the increase (++) and decrease (--) operators have greater operator precedence than the dereference operator (*), but both have a special behavior when used as suffix (the expression is evaluated with the value it had before being increased). Therefore, the following expression may lead to confusion:

```
*p++
```

Because ++ has greater precedence than *, this expression is equivalent to *(p++). Therefore, what it does is to increase the value of p (so it now points to the next element), but because ++ is used as postfix the whole expression is evaluated as the value pointed by the original reference (the address the pointer pointed to before being increased).

Notice the difference with:

```
(*p)++
```

Here, the expression would have been evaluated as the value pointed by p increased by one. The value of p (the pointer itself) would not be modified (what is being modified is what it is being pointed to by this pointer).

If we write:

```
*p++ = *q++;
```

Because ++ has a higher precedence than *, both p and q are increased, but because both increase operators (++) are used as postfix and not prefix, the value assigned to *p is *q **before** both p and q are increased. And then both are increased. It would be roughly equivalent to:

```
*p = *q;
++p;
++q;
```

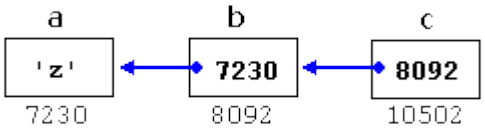
Like always, I recommend you to use parentheses `()` in order to avoid unexpected results and to give more legibility to the code.

2.9.7 Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (`*`) for each level of reference in their declarations:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

This, supposing the randomly chosen memory locations for each variable of `7230`, `8092` and `10502`, could be represented as:



The value of each variable is written inside each cell; under the cells are their respective addresses in memory.

The new thing in this example is variable `c`, which can be used in three different levels of indirection, each one of them would correspond to a different value:

- `c` has type `char**` and a value of `8092`
- `*c` has type `char*` and a value of `7230`
- `**c` has type `char` and a value of `'z'`

2.9.8 void pointers

The `void` type of pointer is a special type of pointer. In C++, `void` represents the absence of type, so void pointers are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereference properties).

This allows void pointers to point to any data type, from an integer value or a float to a string of characters. But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced (which is logical, since we have no type to dereference to), and for that reason we will always have to cast the address in the void pointer to some other pointer type that points to a concrete data type before dereferencing it.

One of its uses may be to pass generic parameters to a function:

```
// increaser
#include <iostream>
```

```
y, 1603
```

```
using namespace std;

void increase (void* data, int psize)
{
  if ( psize == sizeof(char) )
  { char* pchar; pchar=(char*)data;
++(*pchar); }
  else if (psize == sizeof(int) )
  { int* pint; pint=(int*)data; ++(*pint);
}
}

int main ()
{
  char a = 'x';
  int b = 1602;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  cout << a << ", " << b << endl;
  return 0;
}
```

`sizeof` is an operator integrated in the C++ language that returns the size in bytes of its parameter. For non-dynamic data types this value is a constant. Therefore, for example, `sizeof(char)` is `1`, because `char` type is one byte long.

2.9.9 Null pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the integer value zero to any pointer type.

```
int * p;
p = 0;      // p has a null pointer value
```

Do not confuse null pointers with void pointers. A null pointer is a value that any pointer may take to represent that it is pointing to "nowhere", while a void pointer is a special type of pointer that can point to somewhere without a specific type. One refers to the value stored in the pointer itself and the other to the type of data it points to.

2.9.10 Pointers to functions

C++ allows operations with pointers to functions. The typical use of this is for passing a function as an argument to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses `()` and an asterisk (`*`) is inserted before the name:

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

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }
```

```
8
```

```
10
11 int operation (int x, int y, int
12 (*functocall)(int,int))
13 {
14   int g;
15   g = (*functocall)(x,y);
16   return (g);
17 }
18
19 int main ()
20 {
21   int m,n;
22   int (*minus)(int,int) = subtraction;
23
24   m = operation (7, 5, addition);
25   n = operation (20, m, minus);
26   cout <<n;
27   return 0;
   }
```

In the example, `minus` is a pointer to a function that has two parameters of type `int`. It is immediately assigned to point to the function `subtraction`, all in a single line:

```
int (* minus)(int,int) = subtraction;
```

2.10 Virtual Functions

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`:

```
1  // virtual members
2  #include <iostream>
3  using namespace std;
4
5  class CPolygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b; }
11     virtual int area ()
12       { return (0); }
13   };
14
15 class CRectangle: public CPolygon {
16   public:
17     int area ()
18       { return (width * height); }
19   };
20
21 class CTriangle: public CPolygon {
22   public:
23     int area ()
```

```
20
10
0
```

```
24        { return (width * height / 2); }
25    };
26
27  int main () {
28    CRectangle rect;
29    CTriangle trgl;
30    CPolygon poly;
31    CPolygon * ppoly1 = &rect;
32    CPolygon * ppoly2 = &trgl;
33    CPolygon * ppoly3 = &poly;
34    ppoly1->set_values (4,5);
35    ppoly2->set_values (4,5);
36    ppoly3->set_values (4,5);
37    cout << ppoly1->area() << endl;
38    cout << ppoly2->area() << endl;
39    cout << ppoly3->area() << endl;
40    return 0;
41  }
```

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 above example.

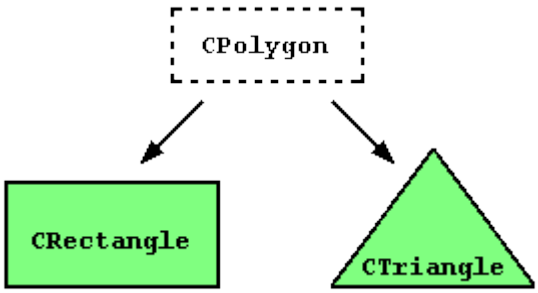
A class that declares or inherits a virtual function is called a *polymorphic class*.

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.

2.10.1 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 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.

```
1  // derived classes
2  #include <iostream>
3  using namespace std;
4
5  class CPolygon {
6    protected:
7      int width, height;
8    public:
9      void set_values (int a, int b)
10       { width=a; height=b;}
11   };
12
```

```
20
10
```

```
13 class CRectangle: public CPolygon {
14   public:
15     int area ()
16       { return (width * height); }
17   };
18
19 class CTriangle: public CPolygon {
20   public:
21     int area ()
22       { return (width * height / 2); }
23   };
24
25 int main () {
26   CRectangle rect;
27   CTriangle trgl;
28   rect.set_values (4,5);
29   trgl.set_values (4,5);
30   cout << rect.area() << endl;
31   cout << trgl.area() << endl;
32   return 0;
33 }
```

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`:

```
1 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 most accessible level the members inherited from the class that follows it (in this case `CPolygon`) will have. 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`.

2.11 Polymorphism

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes:

```
#include <iostream>

using namespace std;


class Shape {
```

```
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
      {
         width = a;
         height = b;
      }
      int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0)
      {
        Shape(a, b);
      }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0)
      {
        Shape(a, b);
      }
      int area ()
      {
         cout << "Rectangle class area :" <<endl;
```

```
         return (width * height / 2);
      }
};
// Main function for the program
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Parent class area
Parent class area
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or**static linkage** - the function call is fixed before the program is executed. This is also sometimes called**early binding** because the area() function is set during the compilation of the program.
But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape {
   protected:
      int width, height;
   public:
      Shape( int a=0, int b=0)
```

```
      {
         width = a;
         height = b;
      }
      virtual int area()
      {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result:

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

2.12 Working with files

C++ provides the following classes to perform output and input of characters to/from files:

- **ofstream:** Stream class to write on files
- **ifstream:** Stream class to read from files
- **fstream:** Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes `istream`, and `ostream`. We have already used objects whose types were these classes: `cin` is an object of class `istream` and `cout` is an object of class `ostream`. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to use `cin` and `cout`, with the only difference that we have to associate these streams with physical files. Let's see an example:

```
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a
file.\n";
  myfile.close();
  return 0;
}
```

```
[file example.txt]
Writing this to a file.
```

This code creates a file called `example.txt` and inserts a sentence into it in the same way we are used to do with `cout`, but using the file stream `myfile` instead.

But let's go step by step:

Open a file

The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to *open a file*. An open file is represented within a program by a stream object (an instantiation of one of these classes, in the previous example this was `myfile`) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function `open()`:

```
open (filename, mode);
```

Where `filename` is a null-terminated character sequence of type `const char *` (the same type that string literals have) representing the name of the file to be opened, and `mode` is an optional parameter with a combination of the following flags:

ios::in	Open for input operations.
ios::out	Open for output operations.
ios::binary	Open in binary mode.
ios::ate	Set the initial position at the end of the file. If this flag is not set to any value, the initial position is the beginning of the file.
ios::app	All output operations are performed at the end of the file, appending the content

	to the current content of the file. This flag can only be used in streams open for output-only operations.
ios::trunc	If the file opened for output operations already existed before, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (`|`). For example, if we want to open the file`example.bin` in binary mode to add data we could do it by the following call to member function `open()`:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each one of the `open()` member functions of the classes `ofstream`, `ifstream` and `fstream` has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For `ifstream` and `ofstream` classes, `ios::in` and `ios::out` are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the `open()` member function.

The default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as *text files*, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream object is generally to open a file, these three classes include a constructor that automatically calls the `open()` member function and has the exact same parameters as this member. Therefore, we could also have declared the previous `myfile` object and conducted the same opening operation in our previous example by writing:

```
ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);
```

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member `is_open()` with no arguments. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:

```
if (myfile.is_open()) { /* ok, proceed with output */ }
```

Closing a file

When we are finished with our input and output operations on a file we shall close it so that its resources become available again. In order to do that we have to call the stream's member function `close()`. This member function takes no parameters, and what it does is to flush the associated buffers and close the file:

```
myfile.close();
```

Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.

In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function `close()`.

Text files

Text file streams are those where we do not include the `ios::binary` flag in their opening mode. These files are designed to store text and thus all values that we input or output from/to them can suffer some formatting transformations, which do not necessarily correspond to their literal binary value.

Data output operations on text files are performed in the same way we operated with `cout`:

```
1  // writing on a text file
2  #include <iostream>
3  #include <fstream>
4  using namespace std;
5
6  int main () {
7    ofstream myfile ("example.txt");
8    if (myfile.is_open())
9    {
10     myfile << "This is a line.\n";
11     myfile << "This is another
12 line.\n";
13     myfile.close();
14   }
```

```
[file example.txt]
This is a line.
This is another line.
```

```
  else cout << "Unable to open
file";
  return 0;
}
```

Data input from a file can also be performed in the same way that we did with `cin`:

```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( getline (myfile,line) )
    {
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open
file";

  return 0;
}
```

```
This is a line.
This is another line.
```

This last example reads a text file and prints out its content on the screen. We have created a while loop that reads the file line by line, using `getline`. The value returned by `getline` is a reference to the stream object itself, which when evaluated as a boolean expression (as in this while-loop) is true if the stream is ready for more operations, and false if either the end of the file has been reached or if some other error occurred.

2.13 Templates

C++ Class Templates are used where we have multiple copies of code for different data types with the same logic. If a set of functions or classes have the same functionality for different data types, they becomes good candidates for being written as Templates.

One good area where this C++ Class Templates are suited can be container classes.

2.13.1 Class templates

C++ Class Templates are used where we have multiple copies of code for different data types with the same logic. If a set of functions or classes have the same functionality for different data types, they becomes good candidates for being written as Templates.

One good area where this C++ Class Templates are suited can be container classes.

Declaring C++ Class Templates:

Declaration of C++ class template should start with the keyword template. A parameter should be included inside angular brackets. The parameter inside the angular brackets, can be either the keyword class or typename. This is followed by the class body declaration with the member data and member functions. The following is the declaration for a sample Queue class.

```
//Sample code snippet for C++ Class Template

template <typename T>

class MyQueue

{

        std::vector<T> data;

        public:

        void Add(T const &d);

        void Remove();

        void Print();

};
```

Defining member functions –

If the functions are defined outside the template class body, they should always be defined with the full template definition. Other conventions of writing the function in C++ class templates are the same as writing normal c++ functions.

```
template <typename T> void MyQueue<T> ::Add(T const &d)

{

 data.push_back(d);
```

```
}


template <typename T> void MyQueue<T>::Remove()

{

data.erase(data.begin( ) + 0,data.begin( ) + 1);

}


template <typename T> void MyQueue<T>::Print()

{

        std::vector <int>::iterator It1;

        It1 = data.begin();

        for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )

        cout << " " << *It1<<endl;

}
```

The Add function adds the data to the end of the vector. The remove function removes the first element. These functionalities make this C++ class Template behave like a normal Queue. The print function prints all the data using the iterator.

Full Program – C++ Class Templates:

```
//C++_Class_Templates.cpp


#include <iostream.h>

#include <vector>


template <typename T>

class MyQueue

{
```

```
std::vector<T> data;

public:

void Add(T const &);

void Remove();

void Print();

};


template <typename T> void MyQueue<T> ::Add(T const &d)

{

data.push_back(d);

}


template <typename T> void MyQueue<T>::Remove()

{

data.erase(data.begin( ) + 0,data.begin( ) + 1);

}


template <typename T> void MyQueue<T>::Print()

{

std::vector <int>::iterator It1;

It1 = data.begin();

for ( It1 = data.begin( ) ; It1 != data.end( ) ; It1++ )

cout << " " << *It1<<endl;


}

//Usage for C++ class templates
```

```
void main()

{

MyQueue<int> q;

q.Add(1);

q.Add(2);



cout<<"Before removing data"<<endl;

q.Print();



q.Remove();

cout<<"After removing data"<<endl;

q.Print();

}
```

Advantages of C++ Class Templates:

One C++ Class Template can handle different types of parameters.
Compiler generates classes for only the used types. If the template is instantiated for int type, compiler generates only an int version for the c++ template class.
Templates reduce the effort on coding for different data types to a single set of code.
Testing and debugging efforts are reduced.

2.13.2 Function templates

Function templates are implemented like regular functions, except they are prefixed with the keyword template. Here is a sample with a function template.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
```

Using Template Functions

Using function templates is very easy: just use them like regular functions. When the compiler sees an instantiation of the function template, for example: the call max(10, 15) in function main, the compiler generates a function max(int, int). Similarly the compiler generates definitions for max(char, char) and max(float, float) in this case.

```
#include <iostream>
using namespace std ;
//max returns the maximum of the two elements
template <class T>
T max(T a, T b)
{
        return a > b ? a : b ;
}
void main()
{

        cout << "max(10, 15) = " << max(10, 15) << endl ;
        cout << "max('k', 's') = " << max('k', 's') << endl ;
        cout << "max(10.1, 15.2) = " << max(10.1, 15.2) << endl ;
}
```

Program Output

```
max(10, 15) = 15
max('k', 's') = s
max(10.1, 15.2) = 15.2
```

2.14 Exception handling

Exceptions are errors that occur at runtime. They are caused by a wide variety of exceptional circumstance, such as running out of memory, not being able to open a file, trying to initialize an object to an impossible value, or using an out-of-bounds index to a vector.
Exceptions provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally and all handlers are ignored.

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword `catch`, which must be placed immediately after the try block:

```
1 // exceptions
2 #include <iostream>
3 using namespace std;
4
```

```
An exception occurred.
Exception Nr. 20
```

```
int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred.
Exception Nr. " << e << endl;
  }
  return 0;
}
```

The code under exception handling is enclosed in a `try` block. In this example this code simply throws an exception:

```
throw 20;
```

A throw expression accepts one parameter (in this case the integer value `20`), which is passed as an argument to the exception handler.

The exception handler is declared with the `catch` keyword. As you can see, it follows immediately the closing brace of the `try` block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

2.15 string manipulation

C++ provides convenient and powerful tools to manipulate strings.

Strings and Basic String Operations

strings are not a built-in data type, but rather a Standard Library facility. Thus, whenever we want to use strings or string manipulation tools, we must provide the appropriate **`#include`** directive, as shown below:

```
#include <string>
using namespace std;     // Or using std::string;
```

We now use **`string`** in a similar way as built-in data types, as shown in the example below, declaring a variable **`name`**:

```
string name;
```

Unlike built-in data types (**`int`**, **`double`**, etc.), when we declare a **`string`** variable without initialization (as in the example above), we *do have* the guarantee that the variable will be

initialized to an empty string — a string containing zero characters.

C++ strings allow you to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cout << "Enter your name: " << flush;
cin >> name;
    // read string until the next separator
    // (space, newline, tab)

    // Or, alternatively:
getline (cin, name);
    // read a whole line into the string name

if (name == "")
{
    cout << "You entered an empty string, "
         << "assigning default\n";
    name = "John";
}
else
{
    cout << "Thank you, " << name
         << "for running this simple program!"
         << endl;
}
```

C++ strings also provide many string manipulation facilities. The simplest string manipulation that we commonly use is concatenation, or addition of strings. In C++, we can use the **+** operator to concatenate (or “add”) two strings, as shown below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
    // result now contains "hello world"
```

Notice that both **`s1`** and **`s2`** remain unchanged! The operation reads the values and produces a result corresponding to the concatenated strings, but doesn't modify any of the two original strings.

The **+=** operator can also be used. In that case, one string is appended to another one:

```
string result;
string s1 = "hello";
    // without the extra space at the end
string s2 = "world";
result = s1;
result += ' ';
    // append a space at the end
result += s2;
```

After execution of the above fragment, **`result`** contains the string **`"hello world"`**.

You can also use two or more **+** operators to concatenate several (more than 2) strings. The example below shows how to create a string that contains the full name from first name and last name (e.g.,**firstname = "John"**, **lastname = "Smith"**, **fullname = "Smith, John"**).

```
string firstname, lastname, fullname;

cout << "First name: ";
getline (cin, firstname);
cout << "Last name: ";
getline (cin, lastname);

fullname = lastname + ", " + firstname;
cout << "Fullname: " << fullname << endl;
```

Of course, we didn't need to do that; we could have printed it with several **<<** operators to concatenate to the output. The example intends to illustrate the use of strings concatenation in situations where you need to store the result, as opposed to simply print it.

Now, let's review this example to have the full name in format **"SMITH, John"**. Since we can only convert characters to upper case, and not strings, we have to handle the string one character at a time. To do that, we use the square brackets, as if we were dealing with an array of characters, or a vector of characters.

For example, we could convert the first character of a string to upper case with the following code:

```
str[0] = toupper (str[0]);
```

The function **toupper** is a Standard Library facility related to character processing; this means that when using it, we have to include the **<cctype>** library header:

```
#include <cctype>
```

If we want to change all of them, we would need to know the length of the string. To this end, strings have a method **length**, that tells us the length of the string (how many characters the string has).

Thus, we could use that method to control a loop that allows us to convert all the characters to upper case:

```
for (string::size_type i = 0; i < str.length(); i++)
{
    str[i] = toupper (str[i]);
}
```

Notice that the subscripts for the individual characters of a string start at zero, and go from **0** to**length-1**.

Notice also the data type for the subscript, **string::size_type**; it is recommended that you always use this data type, provided by the **string** class, and adapted to the particular platform. All string facilities use this data type to represent positions and lengths when dealing with strings.

The example of the full name is slightly different from the one shown above, since we only want to

change the first portion, corresponding to the last name, and we don't want to change the string that holds the last name — only the portion of the full name corresponding to the last name. Thus, we could do the following:

```
fullname = lastname + ", " + firstname;

for (string::size_type i = 0; i < lastname.length(); i++)
{
    fullname[i] = toupper (fullname[i]);
}
```

Search Facilities

Another useful tool when working with strings is the **`find`** method. This can be used to find the position of a character in a string, or the position of a substring. For example, we could find the position of the first space in a string as follows:

```
position = str.find (' ');
```

If the string does not contain any space characters, the result of the find method will be the value**`string::npos`**. The example below illustrates the use of **`string::npos`** combined with the **`find`**method:

```
if (str.find (' ') != string::npos)
{
    cout  << "Contains at least one space!" << endl;
}
else
{
    cout  << "Does not contain any spaces!" << endl;
}
```

The **`find`** methods returns the position of the *first* occurence of the given character (or**`string::npos`**). We also have the related **`rfind`** method — the **`r`** stands for *reverse* search; in other words, **`rfind`** returns the position of the last occurence of the given character, or **`string::npos`**. You could also look at it as the first occurence while starting the search at the end of the string and moving backwards.

The **`find`** and **`rfind`** methods can also be used to find a substring; the following fragment of code can be used to determine if the word **`"the"`** is contained in a given string:

```
string text;
getline (cin, text);

if (text.find ("the") != string::npos)
{
    // ...
```

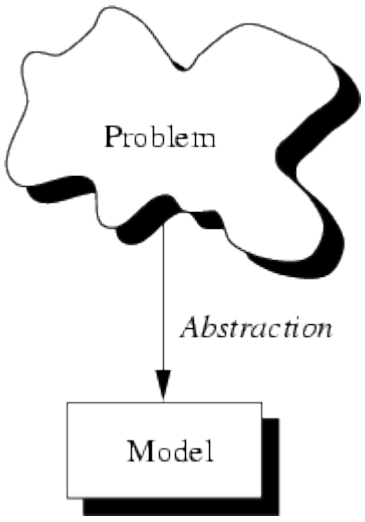
For both cases (searching for a single character or searching for a substring), you can specify a starting position for the search; in that case, the **`find`** method will tell the position of the first occurrence of the search string or character after the position indicated; the **`rfind`** method will return the position of the last occurence before the position inticated — you could look at it as the

first occurrence while moving backwards starting at the specified position. The requirement for this optional parameter is that it must indicate a valid position within the string, which means that the value must be between `0` and `length-1`

2.16 Translating object oriented design into implementations

when writing programs is the *problem*. Typically you are confronted with ``real-life'' problems and you want to make life easier by providing a program for the problem. However, real-life problems are nebulous and the first thing you have to do is to try to understand the problem to separate necessary from unnecessary details: You try to obtain your own abstract view, or *model*, of the problem. This process of modeling is called *abstraction* and is illustrated in Figure :

Figure : Create a model from a problem with abstraction.



The model defines an abstract view to the problem. This implies that the model focusses only on problem related stuff and that you try to define *properties* of the problem. These properties include

- the *data* which are affected and
- the *operations* which are identified

by the problem.

As an example consider the administration of employees in an institution. The head of the administration comes to you and ask you to create a program which allows to administer the employees. Well, this is not very specific. For example, what employee information is needed

by the administration? What tasks should be allowed? Employees are real persons who can be characterized with many properties; very few are:

- name,
- size,
- date of birth,
- shape,
- social number,
- room number,
- hair colour,
- hobbies.

Certainly not all of these properties are necessary to solve the administration problem. Only some of them are *problem specific*. Consequently you create a model of an employee for the problem. This model only implies properties which are needed to fulfill the requirements of the administration, for instance name, date of birth and social number. These properties are called the *data* of the (employee) model. Now you have described real persons with help of an abstract employee.

Of course, the pure description is not enough. There must be some operations defined with which the administration is able to handle the abstract employees. For example, there must be an operation which allows you to create a new employee once a new person enters the institution. Consequently, you have to identify the operations which should be able to be performed on an abstract employee. You also decide to allow access to the employees' data only with associated operations. This allows you to ensure that data elements are always in a proper state. For example you are able to check if a provided date is valid.

To sum up, abstraction is the structuring of a nebulous problem into well-defined entities by defining their data and operations. Consequently, these entities *combine* data and operations. They are **not** decoupled from each other.

Large projects, say, a calendar program, should be split into manageable pieces, often called *modules*. Modules are implemented in separate files and we will now briefly discuss how modularization is done in C and C++. This discussion is based on UNIX and the GNU C++ compiler. If you are using other constellations the following might vary on your side. This is especially important for those who are using integrated development environments (IDEs), for example, Borland C++.

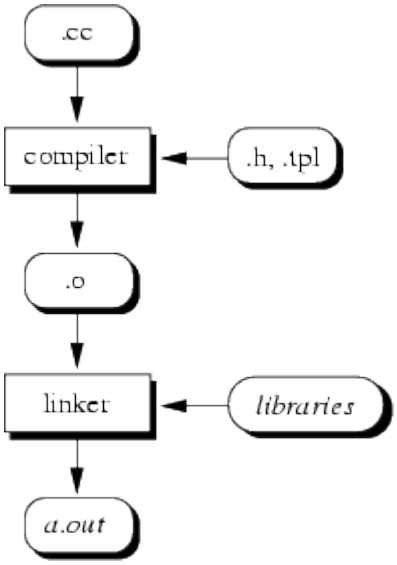
Roughly speaking, modules consist of two file types: *interface descriptions* and *implementation files*. To distinguish these types, a set of suffixes are used when compiling C and C++ programs. Table shows some of them.

Table 9.2: Extensions and file types.

Extension(s)	File Type
`.h, .hxx, .hpp`	interface descriptions ("header" or "include files")
`.c`	implementation files of C
`.cc, .C, .cxx, .cpp, .c++`	implementation files of C++
`.tpl`	interface description (templates)

Compilation Steps

Figure : Compilation steps.



With modern compilers both steps can be combined. For example, our small example programs can be compiled and linked with the GNU C++ compiler as follows (``example.cc" is just an example name, of course):

```
gcc example.cc
```

OOP Term	***Definition***
method	Same as *function*, but the typical OO notation is used for the call, ie, f(x,y) is written x.f(y) where x is an object of class that contains this f method.
send a message	Call a function (method)
instantiate	Allocate a class/struct object (ie, instance) with new
class	A struct with both data and functions
object	Memory allocated to a class/struct. Often allocated with *new*.
member	A field or function is a member of a class if it's defined in that class
constructor	Function-like code that initializes new objects (structs) when they instantiated (allocated with new).
destructor	Function-like code that is called when an object is *deleted* to free any resources (eg, memory) that is has pointers to.
inheritance	Defining a class (child) in terms of another class (parent). All of the public members of the public class are available in the child class.
polymorphism	Defining functions with the same name, but different parameters.
overload	A function is overloaded if there is more than one definition. See polymorphism.
override	Redefine a function from a parent class in a child class.
subclass	Same as child, derived, or inherited class.
superclass	Same as parent or base class.
attribute	Same as data member or member field.