location of the actual parameter. Thus, any access made to reference formal parameters in `swap()` refers to the actual parameters. The statements

```
t = x;
x = y;
y = t;
```

in the body of `swap()` function, internally (as treated by the compiler) have the following meaning,

```
t = *x;        // store the value pointed by x into t
*x = *y;       // store the value pointed by y into location pointed by x
*y = t;        // store the value hold by 't' into location pointed by y
```

because, the formal parameters are of reference type and therefore, the compiler treats them similar to pointers, but does not allow the modification of the address stored in them.

Void Argument List

A function prototype in C with an empty argument list, such as

```
extern void func ();
```

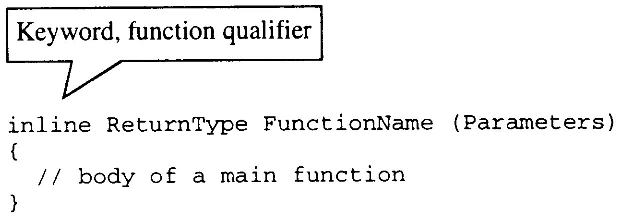
implies that the argument list of the declared function is not prototyped; the compiler will not be able to warn against improper argument usage. To declare a function in C which has no arguments, the keyword `void` is used, as indicated:

```
extern void func (void);
```

In C++, the above two declarations are equivalent. Because C++ maintains strict type checking, an empty argument list is interpreted as the absence of any parameter.

2.11 Inline Functions

Function execution involves the overhead of jumping to and from the calling statement. Trading of this overhead in execution time is considerably large whenever a function is small, and hence in such cases, inline functions can be used. A function in C++ can be treated as a macro if the keyword **`inline`** precedes its definition. The syntax of representing the `inline` function is shown in Figure 2.9.



```
inline ReturnType FunctionName (Parameters)
{
  // body of a main function
}
```

Figure 2.9: Syntax of inline function

Example: An inline function to find square of a number is as follows:

```
inline float square( float x )
{
   x = x * x;
   return( x );
}
```

The significant feature of `inline` functions is that there is no explicit function call; the function body is substituted at the point of inline function call. Thereby, the runtime overhead for function linkage mechanism is reduced. The program `square.cpp` uses an inline function in the computation of the square of a number.

```
// square.cpp: square of a number using inline function
#include <iostream.h>
inline float square( float x )
{
   x = x * x;
   return( x );
}
void main()
{
   float num;
   cout << "Enter a Number <float>: ";
   cin >> num;
   cout << "Its Square = " << square( num );
}
```

Run

```
Enter a Number <float>: 5.5
Its Square = 30.25
```

In `main()`, the statement

```
      cout << "Its Square = " << square( num );
```

invokes the `inline` function `square(..)`. It will be suitably replaced by the instruction(s) of the `square(..)` function body by the compiler. The execution time of the function `square(..)` is less than the time required to establish a linkage between the function *caller* (calling function) and the *callee* (called function). This process involves the operation of saving the actual parameters and function return address onto the stack, followed by a call to the function. On return, the stack must be cleaned to restore the old status. This process is costlier in comparison to having square computation instruction within a program itself instead of a function. Thus, support of `inline` functions allow to enjoy the flexibility and benefits of modular programming, while at the same time delivering computational speedup of macros. Functions having small body do not increase the code size even though they are physically substituted at the point of a call; there is no code for function linkage mechanism. Hence, it is advisable to define functions having small function body as inline functions.

2.12 Function Overloading

A *word* is said to be overloaded when it has two or more distinct meanings. The intended meaning of any particular use is determined by its context. In C++, two or more functions can be given the same name provided each has a unique signature (in either the number or data type of their arguments).

In C++, it is possible to define several functions with the same name, but which perform different actions. It helps in reducing the need for unusual function names, making code easier to read. The functions must only differ in the argument list. For example

```
      swap( int, int );       // prototype
      swap( float, float );   // prototype
```

From a user's view point, there is only one function performing swapping of numbers.

Consider the C program show.c having multiple show() functions for displaying input messages to illustrate the importance of function overloading.

```
/* show.c: display different types of information with different functions */
#include <stdio.h>
void show_integer( int val )
{
    printf ("Integer: %d\n", val);
}
void show_double( double val )
{
    printf ("Double: %lf\n", val);
}
void show_string( char *val )
{
    printf ("String: %s\n", val);
}
int main ()
{
    show_integer( 420 );
    show_double( 3.1415 );
    show_string( "Hello World\n!" );
    return( 0 );
}
```

Run

```
Integer: 420
Double: 3.141500
String: Hello World
!
```

The above program has the following three different functions

```
void show_integer( int val );
void show_double( double val );
void show_string( char *val );
```

performing the same operations, but on different data types. Logically, all the three functions display the value of the input parameters. It has unusual names such as show_integer, show_double, etc., making the task of programming difficult and recalling function names although all of them perform the same operation logically. In C++, this difficulty is circumvented by using the feature of the function name overloading. All the functions performing the same operation must differ in input arguments data-type or in the number of arguments. The program show.cpp equivalent of C's show.c is written using function overloading features.

```
// show.cpp: display different types of information with same function
#include <iostream.h>
void show( int val )
{
    cout << "Integer: " << val << endl;
}
```

```
void show( double val )
{
    cout << "Double: " << val << endl;
}
void show( char *val )
{
    cout << "String: " << val << endl;
}
int main ()
{
    show( 420 );                    // calls show( int val );
    show( 3.1415 );                 // calls show( double val );
    show( "Hello World\n!" );       // calls show( char *val );
    return( 0 );
}
```

Run

```
Integer: 420
Double: 3.1415
String: Hello World
!
```

In the above program, three functions named `show()` are defined, which only differ in their argument lists: `int`, `double`, or `char*`. The functions have the same name. The definition of several functions with the same name is called *function overloading*.

It is interesting to note the way in which the C++ compiler implements function overloading. Although, the functions share the same name in the source text (as in the example above, `show()`), the compiler (and hence the linker) uses different names. The conversion of a name in the source file to an internally used name is called *name mangling*. For instance, the C++ compiler might convert the name `void show(int)` to the internal name `VshowI`, while an analogous function with a `char*` argument might be called `VshowCP`. The actual names which are used internally depend on the compiler and are not relevant to the programmer, except where these names shown in the example, a listing of the contents of a function library.

A few remarks concerning function overloading are the following:

- The usage of more than one function with the same name, but quite different actions should be avoided. In the above example, the functions `show()` are still somewhat related (they print information on the screen). However, it is also quite possible to define two functions, say `lookup()`, one of which would find a name in a list, while the other would determine the video mode. In this case, the two functions have nothing in common except their name. It would therefore be more practical to use names which suggest the action; say, `findname()` and `getvidmode()`.
- C++ does not allow overloaded functions to only differ in their return value. The reason is that processing (testing) of a function return value is always left to the programmer. For instance, the fragment

  ```
      printf ("Hello World!\n");
  ```

  holds no information concerning the return value of the function `printf()` (The return value is, in this case, an integer value that states the number of printed characters. This return value is practically never inspected.). Two functions `printf()` which differ in their return type could therefore, not be distinguished by the compiler.

- Function overloading can lead to surprises. For instance, imagine a usage of a statement such as

  ```
  show( 0 );
  ```

  in the program `show.cpp`; it is difficult to predict which one of the above three `show()` functions is invoked. The zero could be interpreted here as a NULL pointer to a char, i.e., a `(char*) 0`, or as an integer with the value zero. C++ will invoke the function expecting an integer argument, which might not be what one expects.

2.13 Default Arguments

In a C++ function call, when one or more arguments are omitted, the function may be defined to take default values for omitted arguments by providing the default values in the function prototype. These arguments are supplied by the compiler when they are not specified by the programmer explicitly. The program `prnstr.cpp` illustrates the passing of default arguments to function.

```
// prnstr.cpp: default arguments and message printing
#include <iostream.h>
void showstring( char *str = "Hello World!\n" )
{
   cout << str;
}
int main ()
{
   showstring( "Here is an explicit argument\n" );
   showstring();   // in fact this says: showstring ("Hello World!\n");
   return 0;
}
```

Run

```
Here is an explicit argument
Hello World!
```

In `main()`, when the compiler encounters the statement

```
showstring();
```

it is replaced by the statement

```
showstring( "Hello World!\n" );
```

internally. When the function parameter is missing, the compiler substitutes the default parameter in that place.

The possibility of omitting arguments in situations where default arguments are defined is elegant; the compiler will supply the missing arguments, when they are not specified. The code of the program by no means becomes shorter or more efficient. Functions may be defined with more than one default argument.

Default arguments must be known to the compiler prior to the invocation of a function with default arguments. It reduces the burden of passing arguments explicitly at the point of a function call. The program `defarg1.cpp` illustrates the concept of default arguments.

```
// defarg1.cpp: Default arguments to functions
#include <iostream.h>
void PrintLine( char = '-', int = 70 );
void main()
{
   PrintLine();              // uses both default arguments
   PrintLine( '!' );         // assumes 2nd argument as default
   PrintLine( '*', 40 );     // ignores default arguments
   PrintLine( 'R', 55 );     // ignores default arguments
}
void PrintLine( char ch, int RepeatCount )
{
   int i;
   cout << endl;
   for( i = 0; i < RepeatCount; i++ )
      cout << ch;
}
```

Run

```
----------------------------------------------------------------------
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
****************************************
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
```

The feature of default arguments can be utilized to enhance the functionality of the program, without the need for modifying the old code referencing to functions. For instance, the function in the above program

```
void PrintLine( char = '-', int = 70 );
```

prints a line with default character '-' in case it is not passed explicitly. This function can be enhanced to print multiple number of lines, whose new prototype is

```
void PrintLine( char = '-', int = 70, int = 1 );
```

It may be noted that in the new function, the last parameter specifies the number of lines to be printed and by default, it is 1. Therefore, the old code referring to this function need not be modified and new statements can be added without affecting the functionality. The program `defarg2.cpp` has extended the capability of `defarg1.cpp` program.

```
/* defarg2.cpp: Default arguments to functions
             Extending the functionality of defarg1.cpp module */
#include <iostream.h>
void PrintLine( char = '-', int = 70, int = 1 );
void main()
{
   PrintLine();              // uses both default arguments
   PrintLine( '!' );         // assumes 2nd argument as default
   PrintLine( '*', 40 );     // ignores default arguments
   PrintLine( 'R', 55 );     // ignores default arguments
   // new code, Note: old code listed above is unaffected
   PrintLine( '&', 25, 2 );
}
```

```
void PrintLine( char ch, int RepeatCount, int nLines )
{
   int i, j;
   for( j = 0; j < nLines; j++ )
   {
      cout << endl;
      for( i = 0; i < RepeatCount; i++ )
         cout << ch;
   }
}
```

Run

```
------------------------------------------------------------------------
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
****************************************
RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
&&&&&&&&&&&&&&&&&&&&&&&&&
&&&&&&&&&&&&&&&&&&&&&&&&&
```

The following statements in the above two programs

```
PrintLine();                 // uses both default arguments
PrintLine( '!' );            // assumes 2nd argument as default
PrintLine( '*', 40 );        // ignores default arguments
PrintLine( 'R', 55 );        // ignores default arguments
```

are the same. Although, the functionality of the function `PrintLine`, is enhanced in `defarg2.cpp` program, the old code referring to it remains unaffected in terms of its functionality; the compiler supplies the last argument as 1, thereby the new function does the same operation as that of the old one. Thus, default arguments feature can be potentially utilized in extending the fnnction without modifying the old code. Note that all arguments in a multiple argument function need not have default values.

2.14 Keyword `typedef`

The keyword `typedef` is allowed in C++, but no longer necessary, when it is used as a prefix in `enum`, `struct`, or `union` declarations. This is illustrated in the following example:

```
struct somestruct
{
   int a;
   double d;
   char string [80];
};
```

When a `struct`, `enum`, or any other compound type is defined, the tag of this type can be used as type name (`somestruct` is the tag in the above example). For instance, the statement

```
somestruct  what;
```

defines the structure variable `what`. In C, the same variable is defined as

```
struct somestruct  what;
```

Thus, the use of keyword `struct` in the structure variable is default. In C++, the members of the structure variables are accessed similar to C. The statement

```
what.d = 3.1415;
```

assigns the numeric value `3.1415` to `d`, which is a member of the structure variable `what`. The structure declaration and its use in the definition of variables is illustrated in the program `date1.cpp`.

```
// date1.cpp: displaying birth date of the authors
#include <iostream.h>
struct date
{                       //specifies a structure
   int  day;
   int month;
   int year;
};
void main()
{
   date d1 = { 26, 3, 1958 };
   date d2 = { 14, 4, 1971 };
   date d3 = { 1, 9, 1973 };
   cout << "Birth Date of the First Author: ";
   cout << d1.day << "-" << d1.month << "-" << d1.year << endl;
   cout << "Birth Date of the Second Author: ";
   cout << d2.day << "-" << d2.month << "-" << d2.year << endl;
   cout << "Birth Date of the Third Author: ";
   cout << d3.day << "-" << d3.month << "-" << d3.year << endl;
}
```

Run

```
Birth Date of the First Author: 26-3-1958
Birth Date of the Second Author: 14-4-1971
Birth Date of the Third Author: 1-9-1973
```

2.15 Functions as a Part of a `struct`

Structures in C++ have undergone major revisions. Like C structures, C++ structures also provide a mechanism to group together data of different types, into one unit belonging to the same family. In addition to this, C++ allows to associate functions as a part of a structure. Thus, C++ structures provide a true mechanism to handle data abstraction. This is the first concrete example of the definition of an object, as described previously. An object is a structure containing all involved code and data. The general syntax of the C++ structure is:

```
struct StructureName
{
   public:
      // data and functions
   private:
      // data and functions
   protected:
      // data and functions
};
```

The structure has two types of members: data members and member functions. Functions defined within a structure, operate on any member of the structure. The keywords `public`, `private`, and `protected` are called *access specifiers*. If none of these keywords appear in the structure declaration, *all the members of the structure have public access.* The private and protected members of a structure can be accessed only within the structure. Public members of a structure are accessible to both member functions and its instances (structure variables). Internal functions of a structure are privileged code and they can see all the features of a structure, but external code can see only the public features.

A definition of the structure `point` is given in the code fragment below. In this structure, two `int` data fields and one function `draw()` are declared.

```
struct point
{
   int  x, y;              // coordinates
   void draw (void);       // drawing function
};
```

A similar structure could be a part of the painting program used to represent a pixel in the drawing. The following are the points to be noted about structures:

- The function `draw()`, which occurs in the structure body is only a declaration. The actual code of the function, or in other words, the actions to be performed by the function are located elsewhere in the code section of the program. Member function can also be defined within the body of a structure.
- The size of the structure `point` is just two integers. Though a function is declared in the structure, its size remains unaffected. The compiler implements this behavior by allowing the function `draw()` to be known only in the context of the `point` structure.

The point structure could be used as follows:

```
point a, b;      // two points on the screen
a.x = 0;         // define first dot
a.y = 10;        // and draw it
a.draw ();
b = a;           // copy a to b
b.y = 20;        // redefine y-coordinate
b.draw ();       // and draw it
```

The function `draw()`, which is a part of the structure, is selected in a manner similar to the selection of data fields; i.e., using the field selector operator (`.`) with value structures or `->` with pointers to structures.

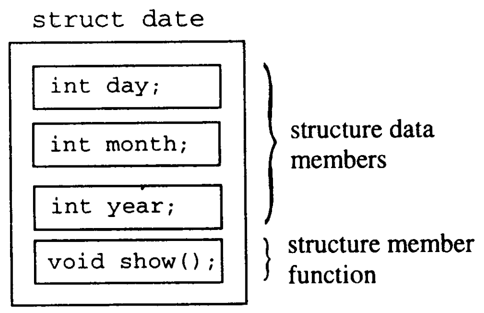


Figure 2.10: Date structure having function show()

The idea behind this syntactical construction is that several structures may contain functions with the same name. For instance, a structure representing a circle might contain three integer values; two values for the coordinates of the center of the circle and one value for the radius. Analogous to the point structure, a `draw()` could be declared in the `circle` structure which would draw the circle.

The program `date2.cpp` is C++ equivalent of the earlier program `date1.cpp`. It illustrates the concept of associating functions operating on structure members as shown in Figure 2.10. The structure `date` has both the data members and functions operating on them. The user accesses the member functions additionally, when compared to C's structure using the *dot operator*.

```
// date2.cpp: displaying birth date of the authors
#include <iostream.h>
struct date
{                      //specifies a structure
   int  day;
   int month;
   int year;
   void show()
   {
      cout << day << "-" << month << "-" << year << endl;
   }
};
void main()
{
   date d1 = { 26, 3, 1958 };
   date d2 = { 14, 4, 1971 };
   date d3 = { 1, 9, 1973 };
   cout << "Birth Date of the First Author: ";
   d1.show();
   cout << "Birth Date of the Second Author: ";
   d2.show();
   cout << "Birth Date of the Third Author: ";
   d3.show();
}
```

Run

```
Birth Date of the First Author: 26-3-1958
Birth Date of the Second Author: 14-4-1971
Birth Date of the Third Author: 1-9-1973
```

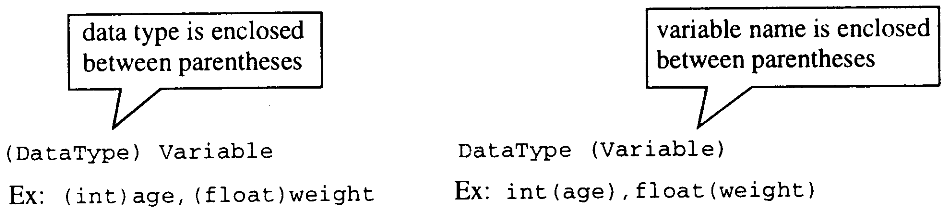
In `main()`, the statements

```
     d1.show();
     d2.show();
     d3.show();
```

invoke the function `show()` defined in the structure `date`.

2.16 Type Conversion

The basic data types can be used with great flexibility in assignments and expressions, due to the implicit type conversion facility provided, whereas with the user-defined data types, the same can be achieved through explicit type conversion (the type cast operator). The syntax of type conversion specification in C and C++ is shown in Figure 2.11.


data type is enclosed between parentheses
variable name is enclosed between parentheses

```
(DataType) Variable
Ex: (int)age, (float)weight
```

```
DataType (Variable)
Ex: int(age), float(weight)
```

(a) Type casting in C **(b) Type casting in C++**

Figure 2.11: Syntax of data type casting in C and C++

Consider the following statements

```
float weight;
int age;
weight = age;
```

where `weight` is of type `float` and `age` is of type `int`. Here, the compiler calls a special routine to convert the contents of `age`, which is represented in an integer format, to a floating-point format, so that it can be assigned to `weight`. The compiler has built-in routines for conversion of basic data types such as char to integer, float to double, etc. The feature of the compiler that performs data conversion without the user intervention, is known as *implicit type conversion.*

The compiler can be instructed explicitly to perform type conversion using the type conversion operators known as type cast operator. For instance, to convert `int` to `float`, the statement is

```
weight = (float) age;
```

where the keyword `float` is enclosed between braces. Here, `float` enclosed between braces is the *type casting operator*. In C++, the above statement can also be expressed in a more readable form as

```
weight = float( age );
```

The explicit conversion of `float` to `int` uses the same built-in routine as implicit conversions. The program `cast.cpp` illustrates the explicit type casting in C++.

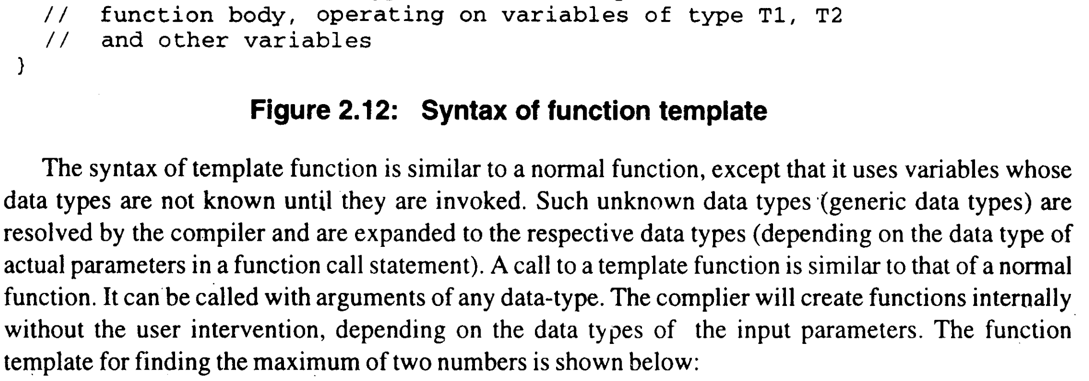
```
// cast.cpp: new style of typecasting in C++
# include <iostream.h>
void main()
{
   int a;
   float b = 420.5;
   cout << "int(10.4) = " << int( 10.4 ) << endl;
   cout << "int(10.99) = " << int( 10.99 ) << endl;
   cout << "b = " << b << endl;
   a = int( b );
   cout << "a = int(b) = " << a << endl;
   b = float( a ) + 1.5;
   cout << "b = float(a)+1.5 = " << b;
```

Run

```
int(10.4) = 10
int(10.99) = 10
b = 420.5
a = int(b) = 420
b = float(a)+1.5 = 421.5
```

2.17 Function Templates

Templates provide a mechanism for creating a single function possessing the capability of several functions, which differ only in their parameters and local variables data type. Such a function is called *function template*. It permits writing one source declaration that can produce multiple functions differing only in their data types. The general format of a template function is depicted in Figure 2.12. A function generated from a function template is known as *template function*, which is created by the compiler internally and is transparent to the user.


Keyword for declaring function template
name of the template data-type
Function parameters of type template, primitive or user-defined

```
template < class T1, class T2, ..>
ReturnType  FunctionName   ( Arguments of type T1 and T2, ... )
{
   // local variables of type T1, T2, or any other
   //  function body, operating on variables of type T1, T2
   //  and other variables
}
```

Figure 2.12: Syntax of function template

The syntax of template function is similar to a normal function, except that it uses variables whose data types are not known until they are invoked. Such unknown data types (generic data types) are resolved by the compiler and are expanded to the respective data types (depending on the data type of actual parameters in a function call statement). A call to a template function is similar to that of a normal function. It can be called with arguments of any data-type. The complier will create functions internally without the user intervention, depending on the data types of the input parameters. The function template for finding the maximum of two numbers is shown below:

```
template <class T>
T max( T a, T b )
{
   if( a > b )
      return a;
   else
      return b;
}
```

The program `mswap.cpp` illustrates the need for function templates. It defines multiple swap functions for swapping the values of different data types.

```
// mswap.cpp: Multiple swap functions
#include <iostream.h>
void swap( char & x, char & y ) // pass by reference
{
   char t;  // temporary used in swapping
   t = x;
   x = y;
   y = t;
}
void swap( int & x, int  & y ) // pass by reference
{
   int t;   // temporary used in swapping
   t = x;
   x = y;
   y = t;
}
void swap( float & x, float & y ) // pass by reference
{
   float t; // temporary used in swapping
   t = x;
   x = y;
   y = t;
}
void main()
{
   char ch1, ch2;
   cout << "Enter two Characters <ch1, ch2>: ";
   cin >> ch1 >> ch2;
   swap( ch1, ch2 ); // compiler calls swap( char &a, char &b );
   cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   swap( a, b );  // compiler calls swap( int &a, int &b );
   cout << "On swapping <a, b>: " << a << " " << b << endl;
   float c, d;
   cout << "Enter two floats <c, d>: ";
   cin >> c >> d;
   swap( c, d );  // compiler calls swap( float &a, float &b );
   cout << "On swapping <c, d>: " << c << " " << d;
}
```

Run

```
Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5
```

The above program has three swap functions

```
void swap( char & x, char  & y );
void swap( int & x, int  & y );
void swap( float & x, float & y );
```

whose logic for swapping is same. Such functions can be defined as template functions without redefining it for every data type. The program gswap.cpp makes all those functions as templates and avoids the overhead of writing the same pattern of code again and again, operating on different data types.

```
// gswap.cpp: generic function for swapping
#include <iostream.h>
template <class T>
void swap( T & x, T & y ) // by reference
{
   T t;  // temporary used in swapping, template variable
   t = x;
   x = y;
   y = t;
}
void main()
{
   char ch1, ch2;
   cout << "Enter two Characters <ch1, ch2>: ";
   cin >> ch1 >> ch2;
   swap( ch1, ch2 );//compiler creates and calls swap( char &a, char &b );
   cout << "On swapping <ch1, ch2>: " << ch1 << " " << ch2 << endl;
   int a, b;
   cout << "Enter two integers <a, b>: ";
   cin >> a >> b;
   swap( a, b );  // compiler creates and calls swap( int &x, int &y );
   cout << "On swapping <a, b>: " << a << " " << b << endl;
   float c, d;
   cout << "Enter two floats <c, d>: ";
   cin >> c >> d;
   swap( c, d );  // compiler creates and calls swap(float &x, float &y );
   cout << "On swapping <c, d>: " << c << " " << d;
}
```

Run

```
Enter two Characters <ch1, ch2>: R K
On swapping <ch1, ch2>: K R
Enter two integers <a, b>: 5 10
On swapping <a, b>: 10 5
Enter two floats <c, d>: 20.5 99.5
On swapping <c, d>: 99.5 20.5
```

In main(), when the compiler encounters the statement

```
swap( ch1, ch2 );
```

calling the swap template function with char type variables, it creates an internal function of type

```
swap( char &a, char &b );
```

The compiler automatically identifies the data type of the arguments passed to the template function, creates a new function, and makes an appropriate call. The process of compiling a template function is totally invisible to the user. Similarly, the compiler translates the following calls

```
swap( a, b );  // compiler creates swap( int &x, int &y );
swap( c, d );  // compiler creates swap( float &x, float &y );
```

into appropriate functions (if necessary), and calls them based on their input parameter data types.

Template Function Overloading

A template function can be overloaded in two ways - (i) by other functions of its name or (ii) by other template functions of the same name. Overloading resolution for functions and template functions can be done in the following three steps:

- If an exact match for the function is found, call it.
- If a function can be generated from a function template matching exactly, then call the generated function.
- If a function can be found by trying ordinary overloading resolution techniques then call it;
- If no match is found, report an error.

2.18 Runtime Memory Management

Whenever an array is defined, a specified amount of memory is set aside at compile time, which may not be utilized fully or may not be sufficient. If a situation arises in which the amount of memory required is unknown at compile time, the memory allocation can be performed during execution. Such a technique of allocating memory during runtime on demand is known as *dynamic memory allocation*.

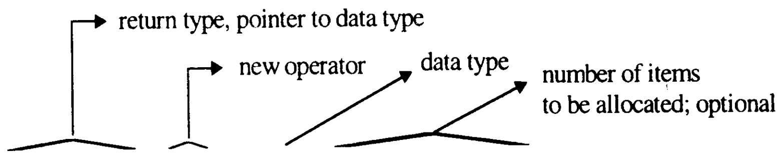
C++ provides the following two special operators to perform memory management dynamically.

- `new` operator for dynamic memory allocation
- `delete` operator for dynamic memory deallocation

The memory management functions such as `malloc()`, `calloc()`, and `free()` in C, have been improved and evolved in C++ as the `new` and `delete` operators to accomplish dynamic memory allocation and deallocation respectively.

`new` Operator

The `new` operator offers dynamic storage allocation similar to the standard library function `malloc`. It is particularly designed keeping OOP in mind and throws an exception if memory allocation fails. The general format of the `new` operator is shown in Figure 2.13.



```
DataType *  new DataType[size in integer];
```

(a) Memory allocation in C++

```
void *malloc (sizeof (DataType) *Size in Integer);
```

(b) Memory allocation in C

Figure 2.13: Syntax of memory allocation in C and C++

The C++ statement

```
PtrVar = new DataType[ IntegerSize ];
```

is equivalent to C's

```
PtrVar = (DataType *) malloc( sizeof( DataType ) * IntegerSize );
```

The operator new allocates a specified amount of memory during runtime and returns a pointer to that memory location. It computes the size of the memory to be allocated by

```
sizeof( DataType ) * IntegerSize
```

where DataType can be a standard data type or a user defined data type. IntegerSize can be an integer expression, which specifies the number of elements in the array. The new operator returns NULL, if memory allocation is unsuccessful.

The following examples illustrate the allocation of memory to various data types.

1.
```
int *a;
a = new int[ 100 ];
```

is equivalent to C's

```
a = (int *) malloc( sizeof( int ) * 100 );
```

It creates a memory space for an array of 100 integers. a[0] will refer to the first element, a[1] to the second element, and so on

2.
```
float *b;
b = new float[ size ];      // size is integer variable
```

is equivalent to

```
b = (float *) malloc( sizeof( float ) * size );
```

3.
```
double *d;
d = new double[ size ];     // size is integer variable
```

is equivalent to

```
d = (double *) malloc( sizeof( double ) * size );
```

4.
```
char *city;
city = new char[ city_name_size ]; // city_name_size is int variable
```

is equivalent to

```
city = (char *) malloc( sizeof( char ) * city_name_size );
```

5.
```
struct date
{                         //specifies a structure
   int  day;
   int month;
   int year;
};
date *date_ptr;
```

The statement

```
date_ptr = new date;
```

is equivalent to

```
date_ptr = (struct date *) malloc( sizeof( date ) );
```

The new operator allows the initialization of memory locations during allocation as follows:

```
PtrVar = new DataType( init_value );
```

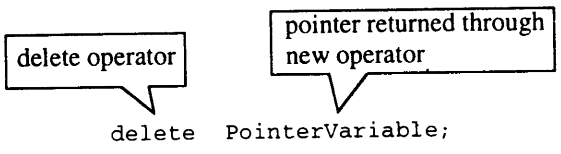
where `init_value` specifies the value to be initialized to a dynamically created element. Note that, `DataType` is optional. It is illustrated by the following examples:

```
int *a = new( 100 );
float *rate = new( 5.5 );
```

The first statement creates a memory for an integer and initializes it with `100` and the second statement creates a memory location for float and initializes it with `5.5`.

`delete` Operator

The `new` operator's counterpart, `delete`, ensures the safe and efficient use of memory. This operator is used to return the memory allocated by the `new` operator back to the memory pool. Memory thus released, will be reused by other parts of the program. Although, the memory allocated is returned automatically to the system, when the program terminates, it is safer to use this operator explicitly within the pointer. This is absolutely necessary in situations where local variables pointing to the memory get destroyed when the function terminates, leaving memory inaccessible to the rest of the program. The syntax of the `delete` operator is shown in Figure 2.14.



```
delete PointerVariable;
```

(a) Memory deallocation in C++

```
free (PointerVariable);
```

(b) Memory deallocation in C

Figure 2.14: Syntax of memory deallocation in C and C++

The C++ statement

```
delete PtrVar;
```

is equivalent to C's

```
free( PtrVar );
```

where `PtrVar` holds the pointer returned by the memory allocation functions such as `new` operator and `malloc()` function. The memory allocated using the `new` operator or `malloc()` function should be released by the `delete` operator and `free()` function respectively.

It should be noted that, by deallocating the memory, the pointer variable does not get deleted and the address value stored in it does not change. However, this address becomes invalid, as the returned memory will be used up for storing entirely different data.

The following examples illustrate the use of the `delete` operator in releasing memory allocated in the earlier memory allocation examples.

1. `delete a;`

is equivalent to C's

```
free( (int *) a );
```

2. `delete b;`

is equivalent to

```
free( (float *) b );
```

3. `delete d;`

is equivalent to

```
free( (double *) d );
```

4. `delete city;`

is equivalent to

```
free( (char *) city );
```

5. `delete date_ptr;`

is equivalent to

```
free( (struct date *) date_ptr );
```

The program `vector.cpp` illustrates the concept of dynamic allocation and deallocation using `new` and `delete` operators.

```
// vector.cpp: addition of two vectors
#include <iostream.h>
void AddVectors( int *a, int *b, int *c, int size )
{
   for( int i = 0; i < size; i++ )
      c[i] = a[i] + b[i];
}
void ReadVector( int *vector, int size )
{
   for( int i = 0; i < size; i++ )
      cin >> vector[i];
}
void ShowVector( int *vector, int size )
{
   for( int i = 0; i < size; i++ )
      cout << vector[i] << " ";
}
void main()
{
   int vec_size;
   int *x, *y, *z;
   cout << "Enter Size of Vector: ";
   cin >> vec_size;
   // allocate memory for all the three vectors
   x = new int[ vec_size ];   // x becomes array of size vec_size
   y = new int[ vec_size ];   // y becomes array of size vec_size
   z = new int[ vec_size ];   // z becomes array of size vec_size
   cout << "Enter elements of vector x: ";
   ReadVector( x, vec_size );
   cout << "Enter elements of vector y: ";
   ReadVector( y, vec_size );
   AddVectors( x, y, z, vec_size ); // z = x+y
```

```
    cout << "Summation Vector z = x + y: ";
    ShowVector( z, vec_size );
    // free memory allocated to all the three vectors
    delete x;    // memory allocated to x is released
    delete y;    // memory allocated to y is released
    delete z;    // memory allocated to z is released
}
```

Run

```
Enter Size of Vector: 5
Enter elements of vector x: 1 2 3 4 5
Enter elements of vector y: 2 3 1 0 4
Summation Vector z = x + y: 3 5 4 4 9
```

In `main()`, the following statements

```
x = new int[ vec_size ];    // x becomes array of size vec_size
y = new int[ vec_size ];    // y becomes array of size vec_size
z = new int[ vec_size ];    // z becomes array of size vec_size
```

allocate memory of size `vec_size` (integer value read previously) to the integer pointer variables x, y, and z respectively. It is equivalent to defining an array of size `vec_size` statically but the size of the array must be known at compile time. This inflexibility of array definition is circumvented by using dynamic allocation known as programmer-controlled memory management. The following statements

```
delete x;    // memory allocated to x is released
delete y;    // memory allocated to y is released
delete z;    // memory allocated to z is released
```

release the memory of size `vec_size` (integer value read previously) allocated to the integer pointer variables x, y, and z respectively. An array defined statically is released automatically by the system whenever the array goes out of scope. But dynamically allocated arrays must be explicitly released by the `delete` operator.

Comments

Most of the concepts introduced in this chapter serve as a quick introduction to enhancements made to C++ language apart from another notable enhancement that is object-oriented programming support. All the material covered in this chapter are discussed in detail in later relevant chapters. This chapter is mainly aimed at those who are familiar with C and want a quick introduction to C++ language. It allows them to extrapolate from the material in this chapter and similarly from the next chapter (*C++ at a Glance*) to their own programming needs. Beginners should supplement it by writing small; similar programs of their own. Both groups can use this and the next chapter as a frame to hang on to the more detailed descriptions that begin in Chapter 4.

Review Questions

2.1 What are the enhancements added to C++ apart from the object-oriented features ?

2.2 Compare the traditional beginner's Hello World program written in C and C++.

2.3 List the compilers supporting C++. Explain their compilation features.

2.4 In C/C++, why is the `main()` function popularly called as the driver function ?

2.5 Enumerate the important features of stream-based I/O and provide a comparative analysis with its

C counterpart statements such as `scanf()` and `printf()`.

2.6 Write an interactive program for computing the roots of a quadratic equation by handling all possible cases. Use streams to perform I/O operations.

2.7 What are the benefits of commenting a program ? Develop a program to illustrate how commenting helps in writing a program, which can be understood by others easily ?

2.8 Why are variables defined with `const` called as read-only variables ? What are its benefits when compared to macros ?

2.9 Justify the need of the scope resolution operator for accessing global variables.

2.10 What are the benefits of defining variables at the point of use ? In the following statement:

```
for( int i = 0; i < 10; i++ )
    xxx;
```

is the variable `i` visible after the termination of loop ?

2.11 What are the differences between reference variables and normal variables ? Why cannot a constant value be initialized to variables of reference type ?

2.12 What are the benefits of strict type checking ? Explain with suitable examples.

2.13 What are the different types of parameter passing methods supported in C++ ? Provide a comparative analysis between pass-by-pointer and pass-by-reference methods.

2.14 What is the difference between inline functions and normal functions ? Write an interactive program with an inline function for finding the maximum value of two numbers.

2.15 What is function overloading ? Explain how it helps in writing well thought-out programs.

2.16 What is name mangling and explain its need ? Is this transparent to the user ?

2.17 Write an interactive program for swapping integer, real, and character type variables without using function overloading. Write the same program by using function overloading features and compare the same with its C counterpart.

2.18 Explain the need of default arguments. Write an interactive program for drawing chart of marks scored by a student in different subjects. A default arguments function has to support statements such as:

```
DrawChart( 50 );
DrawChart( 60, '*' );
DrawChart( 34, '?' );
```

By default, `DrawChart()` draws chart by using star symbols.

2.19 What are the improvements made to the `struct` construct in C++ ? What are the benefits of having functions as a part of the structure declaration. Write an interactive program for processing a student record using structures. All functions manipulating structure variable members must be members of that structure.

2.20 Explain the need for type conversion with suitable examples.

2.21 What are function templates ? What are the differences between function template and template function? Write a program to sort numbers using function templates.

2.22 Explain the constructs supported by C++ for runtime memory management. Write an interactive program processing student's results using C++'s memory management operators.

2.23 Write a program for creating variables of the `date` structure dynamically. Can a pointer variable be used to store data in a memory location pointed to by them, with the binding pointer to a specific location.

3

C++ at a Glance

3.1 Introduction

The C++ language evolved as a result of extensions and enhancements to C. It has efficient memory management techniques, provisions for building new concepts, and a new style of program analysis and design. The reason for retaining C as a subset is its popularity among programmers, and moreover, millions of lines of code already written in C can be directly moved to C++ without rewriting. The other advantages are: the syntax and structure of many statements of C closely resemble the actual operation on the computer's internal registers and allow to produce fast executable code.

The most interesting features of C++ are those which support a new style of programming known as object-oriented programming. It emphasizes on data decomposition rather than algorithm decomposition. OOP is generally useful for any kind of application, but it is particularly suited for interactive computer graphics, simulations, databases, artificial intelligence, high-performance computing, and system programming applications. This chapter presents the first impression of C++ with its features of object-oriented programming.

C++ as an object oriented programming language supports modular programming and enables easy maintainability. The most prominent features of C++ that provide a foundation for data abstraction and object-oriented programming are the following:

- Data Encapsulation and Abstraction: Classes
- Inheritance: Derived Class
- Polymorphism: Operator Overloading
- Friend Functions
- Polymorphism: Virtual Functions
- Generic Classes: Class Templates
- Exception Handling
- Streams Computation

3.2 Data Encapsulation and Abstraction—Classes

Data abstraction is the ability to create user-defined data types for modeling real world objects using built-in data types and a set of permitted operators. Encapsulation is achieved by using the `class`, which combines data and functions that operate on the data. Data hiding is achieved by restricting the members of classes as private or protected.

The object oriented programming technique involves the representation of real world problems in terms of objects. C++ provides a new data structure called *class* whose instance is called *object*. A class consists of procedures or methods and data variables.

Class is the basic construct for creating user-defined data types called abstract data types; in a way it supports encapsulation. Encapsulation allows to combine data and functions that operates on them into a single unit. One or more classes grouped together constitute a program. The program `counter1.cpp` illustrates various concepts such as classes and objects, encapsulation, and declaration of abstract data types. The program creates a class with one data member and instantiates two objects to demonstrate the features of classes. It simulates the behavior of an upward counter.

```
// counter1.cpp: counter class having upward counting capability
#include <iostream.h>
class counter
{
  private:
     int value;             // counter value
  public:
     counter()              // No argument constructor
     {
        value = 0;          // initialize counter value to zero
     }
     counter( int val )     // Constructor with one argument
     {
        value = val;        // initialize counter value
     }
     ~counter()             // destructor
     {
        cout << "object destroyed" << endl;
     }
     int GetCounter()       // counter Access
     {
        return value;
     }
     void up()   // increment counter
     {
        value = value + 1;
     }
};
void main()
{
  counter counter1;       // calls no argument constructor
  counter counter2 = 1;   // calls one argument constructor
  cout << "counter1 = " << counter1.GetCounter() << endl;
  cout << "counter2 = " << counter2.GetCounter() << endl;
  // update counters, increment
  counter1.up();
  counter2.up();
  cout << "counter1 = " << counter1.GetCounter() << endl;
  cout << "counter2 = " << counter2.GetCounter() << endl;
}
```

Run

```
counter1 = 0
counter2 = 1
counter1 = 1
```

```
counter2 = 2
object destroyed
object destroyed
```

The following section describes the various parts of the program:

◆ **Class**, encloses the data and functions into a single unit. The name of the class is `counter`. The class `counter` can be used as the user-defined data type for defining its variables called objects.

◆ **Data Members,** describe the data in the abstract data types. The data member in the class counter is `value`. A class can have any number of data members.

◆ **Member Functions,** define the permissible operations of the data type (member variables). The class `counter` has the following member functions:

1. `counter()`	: constructor with no argument
2. `counter(int val)`	: constructor with one argument
3. `~counter()`	: destructor
4. `GetCounter()`	: counter value access interface
5. `up()`	: increment counter

◆ **Constructor,** is a member function having the same name as that of its class and is executed automatically when the class is instantiated (object is created). It is used generally to initialize object data members and allocate the necessary resources to them. The class `counter` has two constructors to initialize the data members of the class.

```
counter()
counter(int)
```

Similar to normal functions, member functions of a class including constructors (but not destructor) differ in their specifications (data types of argument or number of arguments); this feature is called function overloading. The compiler will identify a suitable constructor, whose formal parameters matches with those actual parameters passed to it at the time of creation of objects.

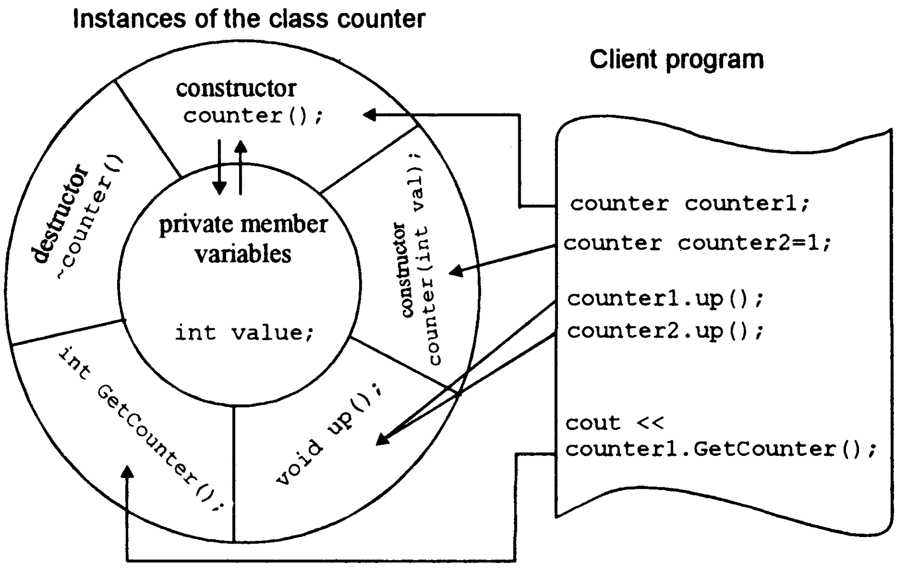
◆ **Destructor,** is a member function having the character ~ (tilde) followed by a function name, which is same as the class name (i.e., `~classname()`) and is invoked automatically when class's object goes out of scope (i.e., the object is no longer needed). It is generally used to reclaim all the resources allocated to the object. The above program has the destructor named `~counter()` in the class `counter`. It is automatically invoked whenever objects go out of scope (when program terminates in the above case). A class can have at the most one destructor.

◆ **Access Specifiers,** control the visibility status of the members of a class. Access specifiers in the above program are the keywords `private` and `public`. The members of the class `counter` declared following the keyword `private` are accessible to only members of its own class. Thus, hiding the data inside a class, so that it is not accessed mistakenly by any function outside the class. Whereas, the members of the class `counter` declared following the keyword `public` are accessible from objects of the class in addition to their own class members.

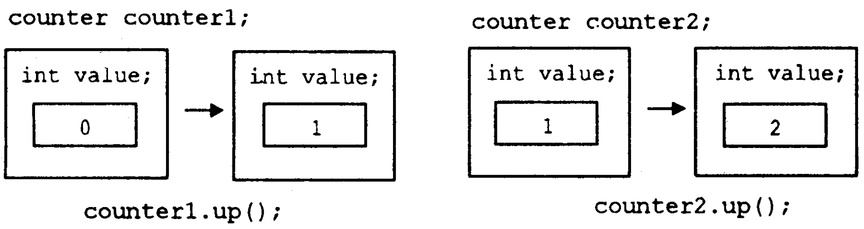
In the above program, the data member `value` is declared as `private` and member functions are declared as `public`. By default, these are `private`. The explicit declaration `public` means that these functions can be accessed from outside the class.

◆ **Object,** is an instance of a class. The objects created in the program are `counter1` and `counter2` which are the instances of the class `counter`. The first object's data member `value` is initialized using zero-argument constructor, whereas the second object is initialized using one-argument constructor.

The pictorial representation of the class `counter` and invocation of its members by various statements in `main()` is shown in the Figure 3.1a.



(a) Counter object and member access



(b) Counter objects status

Figure 3.1: Counter class and objects

In `main()`, the statements

```
counter counter1;          // calls no argument constructor
counter counter2 = 1;      // calls 1 argument constructor
```

create two objects called `counter1` and `counter2` of the class `counter`. The first statement invokes no-argument constructor, `counter()` automatically, which initializes its data member `value` to zero, whereas the second statement invokes a single argument constructor, `counter(int)` automatically and initializes its data member `value` to 1 (as mentioned in the statement). The statements

```
counter1.up();
counter2.up();
```

invoke member function `up()` defined in the class `counter` and increment the data member `value` by one. Thus, the two objects `counter1` and `counter2` of the class `counter` have different data values as shown in Figure 3.1b. Each object of the counter class is stored in a separate area in memory.

Classes are syntactically, an extension of structures. The difference is that, all the members of structures are public by default, whereas members of classes are private by default. Class follows the principle of *all the information about a module should be private to the module unless it is specifically declared public.*

Member Functions

The data members of a class must be declared within the body of a class, whereas the member functions of a class can be defined in one of the following ways:

- Inside the class body
- .Outside the class body

The syntax of a member function definition changes depending on whether it is defined inside or outside the class specification. However, irrespective of the location of its definition (inside or outside the class body), the member function must perform the same operation. Therefore, the code inside the function body would be identical in both the cases. The compiler treats member functions defined inside a class as *inline* functions, whereas those defined outside a class are not treated as inline functions. The program `stdclass.cpp` illustrates the mechanism of defining member functions outside the body of the class.

```
// stdclass.cpp: member functions defined outside a body of the class
#include <iostream.h>
#include <string.h>
class student
{
  private:
     int roll_no;          // roll number
     char name[ 20 ];      // name of a student
  public:
     void setdata( int roll_no_in, char *name_in );
     void outdata();
};
// initializing data members
void student::setdata( int roll_no_in, char *name_in )
{
   roll_no = roll_no_in;
   strcpy( name, name_in );
}
// display data members on the console screen
void student::outdata()
{
   cout << "Roll No = " << roll_no << endl;
   cout << "Name = " << name << endl;
}
void main()
{
   student s1;    // first object/variable of class student
   student s2;    // second object/variable of class student
   s1.setdata( 1, "Tejaswi" );   // object s1 calls member function setdata
   s2.setdata( 10, "Rajkumar" ); // calls member function setdata
   cout << "Student details..." << endl;
   s1.outdata();      // object s1 calls member function outdata
   s2.outdata();      // object s2 calls member function outdata
}
```

Run

```
Student details...
Roll No = 1
Name = Tejaswi
Roll No = 10
Name = Rajkumar
```

In the class `student`, the prototype of member functions `setdata` and `outdata` are declared within the body of the class and they are defined outside the body of the class. In the declarator

```
        void student::outdata()
```

`student::` indicates that the function `outdata()`, belongs to the class `student` and it is a member function of the class `student`.

3.3 Inheritance–Derived Classes

Inheritance is a technique of organizing information in the hierarchical form. It is similar to a child inheriting the features such as beauty of the mother and intelligence of the father. It is an important feature of object oriented programming that allows to extend and reuse existing code without requiring to rewrite it from scratch. Inheritance involves derivation of new classes from the existing ones, thus enabling the creation of a hierarchy of classes, similar to the concepts of class and subclass in the real world. A new class created using an existing class is called the derived class. This process is called inheritance. The derived class inherits the members - both data and functions of the base class. It can also modify or add to the members of a base class. Inheritance allows a hierarchy of classes to be derived.

Derived classes, inherit data members and member functions from their base classes, and can be enhanced by adding other data members and member functions.

Recall that the program `counter1.cpp` discussed above, uses the class `counter` as a general purpose counter variable. A counter could be incremented or decremented. The `counter` class can be extended to support downward counting. It can be achieved by either modifying the `counter` class or by deriving a new class called `NewCounter` from the `counter` class. The program `counter2.cpp` is an extended version of the previous program and has two classes, one, `counter` as a base class and two, `NewCounter` as a derived class. The private members of a base class cannot be inherited.

C++ supports another access specifier called `protected`. Its access privileges are similar to `private` except that they are accessible to its derived classes. Protected access privilege is used when members in base class's section are to be treated as *private* and they must be inheritable by a derived class. The public members of the base class are accessible to the derived class, but the private members of the base class are not. However, the *protected members* of the base class are accessible to the derived class, but they are private to all other classes.

```
// counter2.cpp: new counter having upward and downward counting capability
#include <iostream.h>
class counter
{
  protected:                 // Note: it is private in COUNTER1.CPP
     int value;              // counter value
  public:
     counter()               // No argument constructor
     {
        value = 0;           // initialize counter value to zero
     }
     counter( int val )      // Constructor with one argument
     {
        value = val;         // initialize counter value
     }
     int GetCounter()        // counter Access
     {
        return value;
     }
     void up()      // increment counter
     {
        value = value + 1;
     }
};
// NewCounter is derived from the old class counter publically
class NewCounter: public counter
{
  public:
     NewCounter(): counter()
     {}
     NewCounter( int val ) : counter( val )
     {}
     void down()    // decrement counter
     {
        value = value - 1;   // decrement counter
     }
};
void main()
{
  NewCounter counter1;       // calls no argument constructor
  NewCounter counter2 = 1;   // calls 1 argument constructor
  cout << "counter1 initially = " << counter1.GetCounter() << endl;
  cout << "counter2 initially = " << counter2.GetCounter() << endl;
  // increment counter
  counter1.up();
  counter2.up();
  cout << "counter1 on increment = " << counter1.GetCounter() << endl;
  cout << "counter2 on increment = " << counter2.GetCounter() << endl;
  // decrement counter
  counter1.down();
```

```
   counter2.down();
   cout << "counter1 on decrement = " << counter1.GetCounter() << endl;
   cout << "counter2 on decrement = " << counter2.GetCounter();
}
```

Run

```
counter1 initially = 0
counter2 initially = 1
counter1 on increment = 1
counter2 on increment = 2
counter1 on decrement = 0
counter2 on decrement = 1
```

In the above program, the NewCounter class has its own features to perform counter decrement by using the member functions of the counter. The statement

```
class NewCounter: public counter
```

derives a new class NewCounter known as derived class from the base class counter. The base class counter is publicly inherited by the derived class NewCounter. Hence, the members of counter class that are protected become protected and public become public in the derived class NewCounter. The NewCounter class can treat all the members of the counter class, as though they belong to it.

When an object of the derived class is created, one of the constructors of the base class must be executed before a constructor of the derived class is executed. In the case of destructors, the body of the derived class destructor is executed first followed by that of the base class. The specification of the constructors in the following statements

```
NewCounter(): counter()
NewCounter( int val ) : counter( val )
```

indicate as to which one of the constructors in the base class has to be selected while creating objects of the derived class. If no explicit specification of the base class constructor is made in the derived class constructor, the compiler will select the no-argument constructor of the base class by default as indicated in Figure 3.2.

In main(), the statements

```
NewCounter counter1;      // calls no argument constructor
NewCounter counter2 = 1;  // calls 1 argument constructor
```

create two objects called counter1 and counter2 of the NewCounter class. The first statement invokes the no-argument (default) constructor NewCounter() automatically, which in turn calls the base class constructor counter() to initialize the data member value to zero. Whereas, the second statement invokes the one-argument constructor NewCounter(int) automatically, which in turn calls the base class constructor counter(int) to initialize the data member value to 1 (as mentioned in the statement). Derived class can also initialize its own data members or base class data members explicitly.

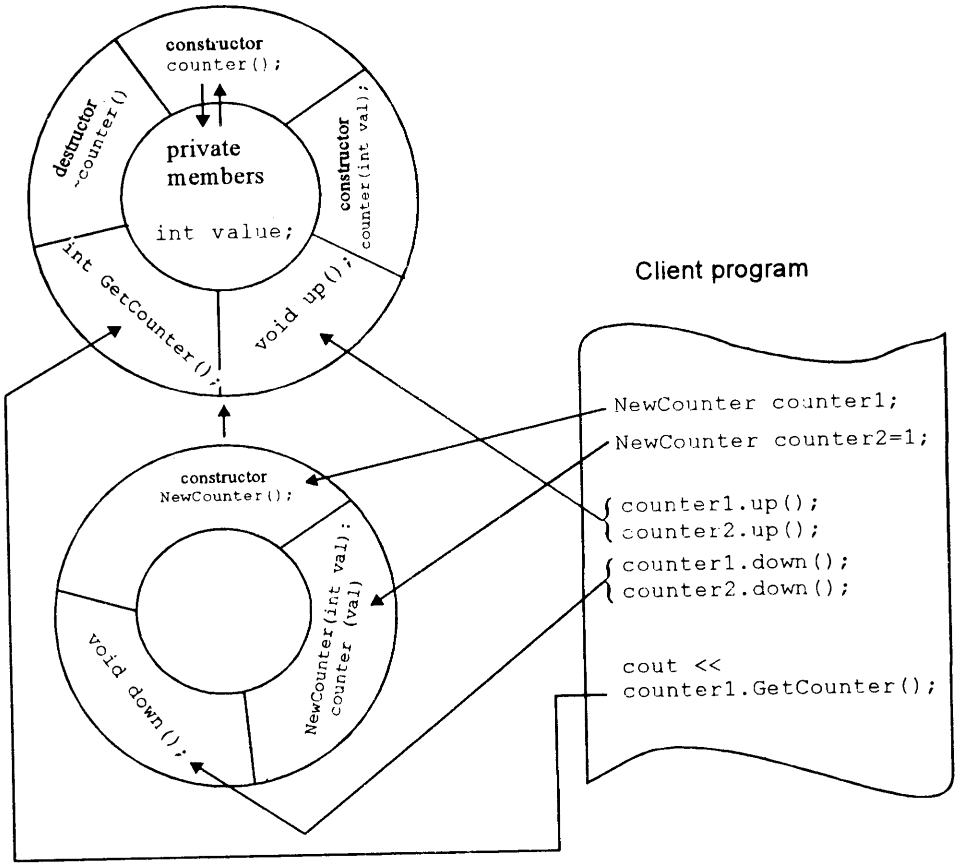
The statements

```
counter1.up();
counter2.up();
```

call member function up() of the base class to increment the counter value by one. Whereas the statements

```
counter1.down();
counter2.down();
```

call member function down() of the derived class to decrement the counter value by one. C++ supports derivation of a class from more than one base class, which is called multiple inheritance. Some of the other forms of inheritance supported by C++ are hierarchical, multilevel, hybrid, and multipath.



```
NewCounter counter1;
NewCounter counter2=1;
counter1.up();
counter2.up();
counter1.down();
counter2.down();

cout <<
counter1.GetCounter();
```

Instances of the class NewCounter

Figure 3.2: NewCounter class and inheritance

3.4 Polymorphism–Operator Overloading

Polymorphism allows a single name/operator to be associated with different operations depending on the type of data passed. In C++, it is realized by using function overloading, operator overloading, and dynamic binding. The operators such as +,-,*,/ etc., dealing with basic data types can be extended to work on user-defined data types by using the facility of operator overloading. Overloaded operators work with user-defined or basic-data types depending upon the type of operands. Operator overloading allows the user to give additional meaning to most operators so that it can be used with the user's own data types, thereby making the data-types easier to use.

Operator overloading, similar to function name overloading, helps to reduce the need for unusual function names, making code easier to understand. It also supports programmer-controlled automatic type conversion, which blend user defined data types, appear and work in the same way as fundamental data types provided by the C++ language.

Operator overloading extends the *semantics* of an operator without changing their *syntax*. The grammatical rules defined by the C++ that govern its use such as the number of operands, precedence, and associativity of the operator remains the same for overloaded operators. Therefore, it should be remembered that overloading of an operator does not change its original meaning. C++ allows overloading of both unary and binary operators.

In the program `counter1.cpp` and `counter2.cpp`, the functions `up()` and `down()` are invoked explicitly to update the counters. Instead of using such functions, the operators like ++ (increment operator) can be used to perform the same job, while increasing the program readability without the loss of functionality. The enhanced version of the class `counter` declared in the program `counter2.cpp` is rewritten to use overloaded increment operator in the program `counter3.cpp`. It overloads increment and decrement operators to operate on user defined data items.

```
// counter3.cpp: increment and decrement operation by operator overloading
#include <iostream.h>
class counter
{
   private:
      int value;              // counter value
   public:
      counter()               // No argument constructor
      {
         value = 0;           // initialize counter value to zero
      }
      counter( int val )      // Constructor with one argument
      {
         value = val;         // initialize counter value
      }
      int GetCounter()        // counter Access
      {
         return value;
      }
      // overloading increment operator
      void operator++()    // increment counter
      {
         value = value + 1;
      }
      void operator --()      // decrement counter
      {
         value = value - 1;   // decrement counter
      }
};
void main()
{
   counter counter1;       // calls no argument constructor
```

```
    counter counter2 = 1;   // calls 1 argument constructor
    cout << "counter1 initially = " << counter1.GetCounter() << endl;
    cout << "counter2 initially = " << counter2.GetCounter() << endl;
    // increment counter
    ++counter1;
    counter2++;
    cout << "counter1 on increment = " << counter1.GetCounter() << endl;
    cout << "counter2 on increment = " << counter2.GetCounter() << endl;
    // decrement counter
    --counter1;
    counter2--;
    cout << "counter1 on decrement = " << counter1.GetCounter() << endl;
    cout << "counter2 on decrement = " << counter2.GetCounter();
}
```

Run

```
counter1 initially = 0
counter2 initially = 1
counter1 on increment = 1
counter2 on increment = 2
counter1 on decrement = 0
counter2 on decrement = 1
```

The word `operator` is a keyword. It is preceded by the return type `void`. The operator to be overloaded is immediately written after the keyword `operator`, followed by the `void` function symbol as `operator++()`. This declarator syntax informs the compiler to call this member function whenever the `++` operator is encountered, provided its operand is of type `counter`.

The statement in the class `counter`

```
    void operator ++()       // increment counter
```

overloads the increment operator (++) to operate on the user defined data type. When the compiler encounters statements such as

```
    ++counter1;
    counter2++;
```

it calls the overloaded operator function defined in the user-defined class (see Figure 3.3). The statement in the class `counter`

```
    void operator--()       // decrement counter
```

overloads the decrement operator (--) to operate on objects of the user defined data type. When the compiler encounters statements such as

```
    --counter1;
    counter2--;
```

it calls the overloaded operator function defined in the user-defined class. It can be observed that the function body of an overloaded and a non-overloaded operator function is same; the only change is in the function prototype and method of calling. For instance, the statement in `counter2.cpp`

```
    counter2.up();
```

can be replaced by a more readable equivalent statement:

```
    counter2++;
```

in the above program.

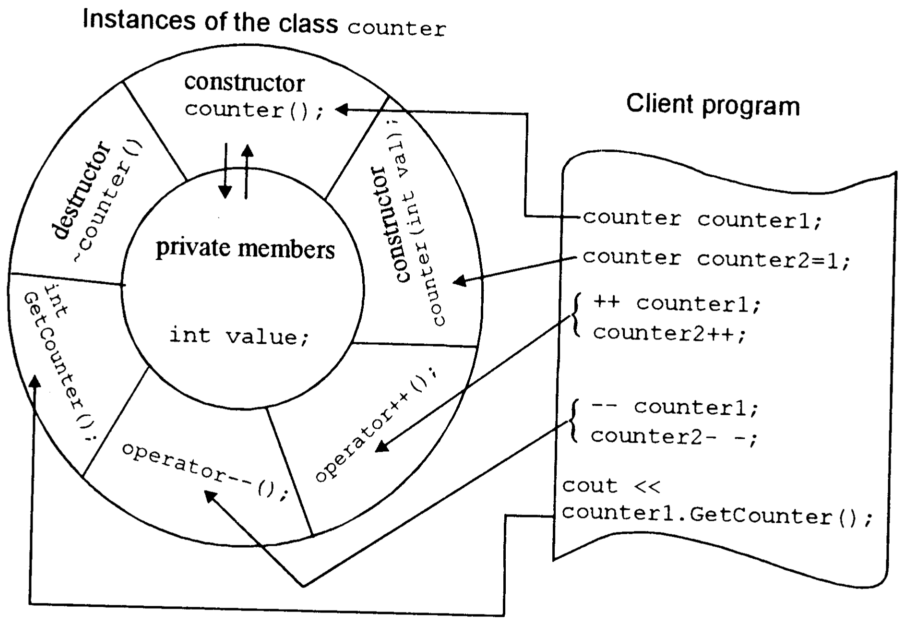


Figure 3.3: Unary operator overloading in counter class

The concept of unary operator overloading also applies equally to binary operators. Addition of two counters without using operator overloading can be performed by a statement such as

```
counter3 = counter1.AddCounter( counter2 );
```

It invokes the member function `AddCounter()` of `counter1` object's class. By overloading the + operator, the above clumsy and dense-looking expression can be represented in a readable and simplified form as:

```
counter3 = counter1 + counter2;
```

A detailed discussion on operator overloading can be found in the chapter on *Operator Overloading*.

3.5 Friend Functions

C++ provides the concept of a *friend class* whose member functions can access the private members of another class. A *friend function* accesses the private data variables of another class. The major difference between an ordinary class function and a friend function is that the ordinary function accesses the object that involves the member function, while a friend function requires objects to be passed by reference or value.

Friend functions play a very important role in operator overloading by providing the flexibility, which is denied by the member functions of a class. It allows overloading of stream operators (<< or >>) for stream computation on user defined data types. The only difference between the friend function and member function is that, the friend function requires all formal arguments to be specified explicitly, whereas the member function takes first formal argument implicitly and the remaining arguments (if any) explicitly. Friend functions can either be used with a unary or binary operator.

Similar to the built-in variables, the user-defined objects can also be read or output using the stream operators: insertion and extraction operators. In the case of the overloaded << operator, the *ostream &* is taken as the first argument of a friend function of a class. The return value of this friend function is of type *ostream &*. Similarly, for overloading the >> operator, the *istream &* is taken as the first argument of a friend function of a class. The return value of this friend function is of type *istream &*. In both the cases, a reference to an object of the current class is taken as a second argument and after storing the result in its second object, its first argument, the istream object would be returned.

The program `counter4.cpp` illustrates the flexibility of overloading the output stream operators and their usage with the user defined objects.

```
// counter4.cpp: overloading stream operator cout << value
#include <iostream.h>
class counter
{
   private:
      int value;            // counter value
   public:
      counter()             // No argument constructor
      {
         value = 0;         // initialize counter value to zero
      }
      counter( int val )    // Constructor with one argument
      {
         value = val;       // initialize counter value
      }
      int GetCounter()      // counter Access
      {
         return value;
      }
      // overloading increment operator
      void operator++()    // increment counter
      {
         value = value + 1;
      }
      // overloading decrement operator
      void operator --()    // decrement counter
      {
         value = value - 1;  // decrement counter
      }
      // overloading binary operator
      counter operator +( counter counter2 );
      friend ostream & operator << ( ostream & Out, counter & counter );
};
// operator function defined outside the class body, hence use :: operator
counter counter::operator +( counter counter2 )
{
   counter temp;
   // value belongs to counter1 and counter2.value is of counter2
   temp.value = value + counter2.value;
   return temp;
}
// it is just a friend function, it is not a member of counter classes
ostream & operator << ( ostream & Out, counter & counter )
{
   // display all internal data of counter class
   cout << counter.value;
   // return output stream Out for cascading purpose
   return Out;
}
void main()
{
   counter counter1;        // calls no argument constructor
   counter counter2 = 1 ;  // calls 1 argument constructor
   cout << "counter1 initially = " << counter1 << endl;
   cout << "counter2 initially = " << counter2 << endl;
   // increment counter
   ++counter1;
   counter2++;
   cout << "counter1 on increment = " << counter1 << endl;
   cout << "counter2 on increment = " << counter2 << endl;
   // decrement counter
   --counter1;
   counter2--;
   cout << "counter1 on decrement = " << counter1 << endl;
   cout << "counter2 on decrement = " << counter2 << endl;
   counter counter3;        // calls no argument constructor
   counter3 = counter1 + counter2;  // calls operator+(counter)
   cout << "counter3 = counter1+counter2 = " << counter3;
}
```

Run

```
counter1 initially = 0
counter2 initially = 1
counter1 on increment = 1
counter2 on increment = 2
counter1 on decrement = 0
counter2 on decrement = 1
counter3 = counter1+counter2 = 1
```

The contents of the object `counter1` can be displayed by using the statement

```
      cout << counter1;
```

instead of using the statement

```
      cout << counter.GetCounter();
```

This is the same as the use of the stream operator to display the contents of variables of standard data type. The operator member function

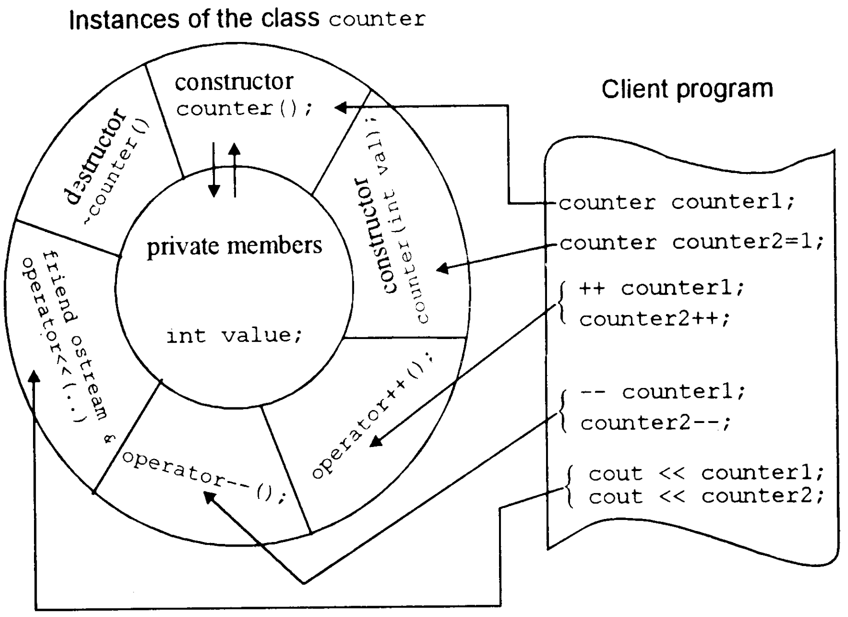
```
      ostream & operator << ( ostream & Out, counter & counter );
```

defined in the `counter` class displays the contents of the objects of the `counter` class (see Figure 3.4). The stream classes, *istream* and *ostream* are declared in the `iostream.h` header file.

The input stream operator can also be overloaded to read objects of the `counter` class, whose prototype can be:

```
istream & operator >> ( istream & In, counter & counter );
```

Note that C++ does not allow overloading of operators =, ( ), [ ], and -> as friend operator functions. however, they can be overloaded as member operator functions.



```
counter counter1;
counter counter2=1;
++ counter1;
counter2++;
-- counter1;
counter2--;
cout << counter1;
cout << counter2;
```

Figure 3.4: Operator overloading and friend functions

3.6 Polymorphism–Virtual Functions

In C++, runtime polymorphism is achieved using virtual functions. Virtual functions facilitate dynamic binding of functions to the appropriate objects. They are the means by which functions of the base class can be overridden by functions of the derived class.

Virtual functions allow derived class to redefine member functions inherited from a base class. General programs can then be written that are obvious to the classes of the objects they manipulate, through dynamic binding. The runtime system will choose the function appropriate to a particular class.

Virtual functions allow programmers to declare functions in a base class that can be redefined in each derived class. When a pointer to the base class is used with a base or derived class object, the object to which it points determines the activation of an appropriate member function call. That is, when a base class pointer points to the object of a derived class, the derived class's member function is selected and when it points to the object of the base class, the base class's member function is selected at runtime.

In C++, calls to virtual member functions are linked at runtime, as a result of which an object's behavior is determined only at runtime. This binding procedure is termed as *late binding*. The keyword `virtual` instructs the compiler that the calls to these member functions are to be linked only at run time. Thus, the choice of member function to be executed depends on the object of a class, the pointer is addressing at runtime. The program `virtual.cpp` illustrates the concept of virtual functions.

```
// virtual.cpp: Binding pointer to base class's object to base or derived
// objects at runtime and invoking respective members if they are virtual
#include <iostream.h>
class Father
{
   protected:
      int f_age;
   public:
      Father( int n )
      {
         f_age = n;
      }
      virtual int GetAge(void)
      {
         return f_age;
      }
};
// Son inherits all the properties of father
class Son : public Father
{
   protected:
      int s_age;
   public:
      Son( int n, int m ):Father(n)
      {
         s_age = m;
      }
      int GetAge(void)
      {
         return s_age;
      }
};
void main()
{
   Father *basep;
   basep = new Father(45);    // pointer to father
   cout << "Father's Age: ";
   cout << basep->GetAge() << endl;  // calls father::GetAge
   delete basep;
   basep = new Son(45, 20);   // pointer to son
   cout << "Son's Age: ";
   cout << basep->GetAge() << endl;  // calls son::GetAge()
   delete basep;
}
```

Run

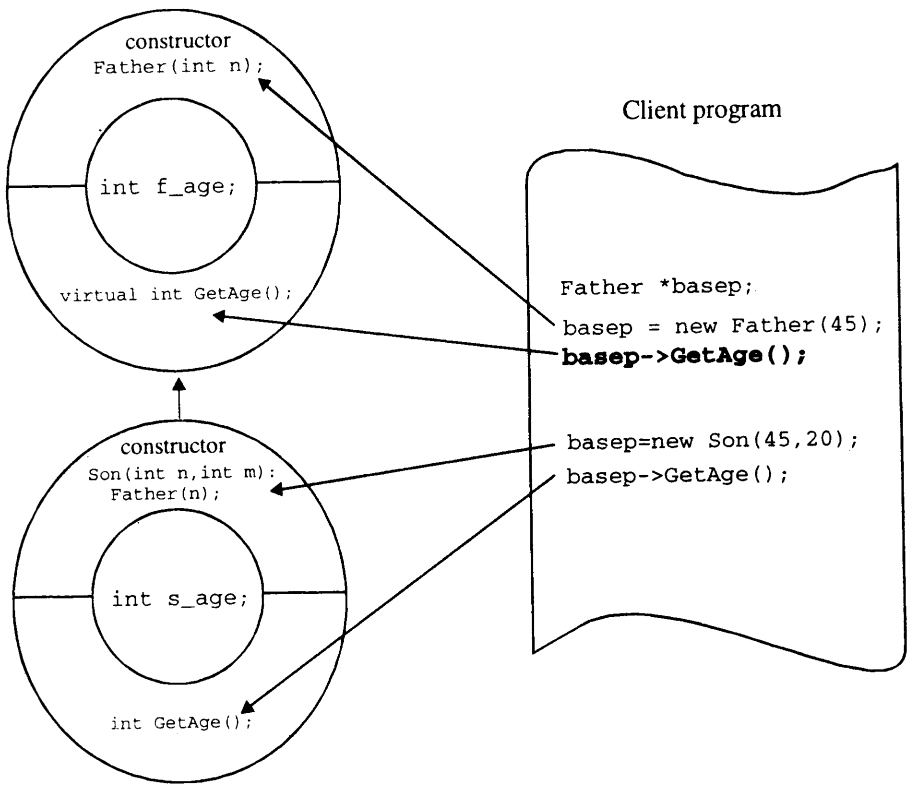
```
Father's Age: 45
Son's Age: 20
```

In the base class `Father`, the statement

```
virtual int GetAge(void)
```

indicates that, an invocation of `GetAge()` through the pointer to an object must be resolved at runtime based on *which class's object the pointer is pointing to.* A pointer to the object of the base class can be made to point to its derived class.

Instances of the class `Father`



Instances of the class `Son`

Figure 3.5: Virtual functions and dynamic binding (base pointer accessing derived objects)

In `main()`, the statement

```
Father *basep;
```

creates a pointer variable to the object of the base class `Father` and the statement

```
basep = new Father(45);     // pointer to Father
```

creates an object of the class `Father` dynamically and assigns its address to the pointer `basep`. The statement

```
cout << basep->GetAge() << endl;  // calls father::GetAge
```

invokes the member function `GetAge()` of the `Father` class.

Similarly, the statement

```
    basep = new Son(45, 20);    // pointer to son
```

creates an object of type class Son dynamically and assigns its address to the pointer basep. The statement

```
    cout << basep->GetAge() << endl;  // calls Son::GetAge
```

invokes the member function GetAge() of the class Son (see Figure 3.5). If a call to a non-virtual function is made in this case, it will invoke the member function of the base class Father instead of the derived class Son. Note that the same pointer is able to invoke base or derived class's member function depending on the class's object to which it is bound (and this is true only with virtual functions).

It is important to note that, *virtual functions must be accessed through the pointer to a base class.* However, they can be accessed through objects instead of pointers, but note that the runtime polymorphism is achieved only when a virtual function is accessed through the pointer to a base class. Also another important aspect is that, when a function is defined as virtual in the base class and if the same function is redefined in the derived class, that function is also treated as virtual function by default. Only class member functions can be declared as virtual functions. Regular functions and *friend* functions do not qualify as virtual functions.

3.7 Generic Classes–Class Templates

The container class (i.e., a class that holds objects of some other type) is of considerable importance when implementing data structures. The limitation of such classes to hold objects of any particular data type can be overcome by declaring that class as a *template class.* It allows several classes which differ only in the data type of their data members to be declared with a single declaration.

A class template arguments can be of type strings, function names, and constant expressions, in addition to template type arguments. Consider the following class template to illustrate, how the compiler handles creation of objects using the class templates:

```
    template <class T, int size>
    class myclass
    {
       T arr[size];.
       ....
    };
```

When objects of template class are created using the statement such as,

```
    myclass <float,10> new1;
```

the compiler creates the following class:

```
    class myclass
    {
       float arr[10];
       ....
    };
```

Again if a statement such as,

```
    myclass <int, 5> new2;
```

is encountered for creating the object new2, the compiler creates the following class:

```
class myclass
{
    int arr[5];
    ....
};
```

The template declaration of the `vector` class is illustrated in the program `vector.cpp`. It has a data member which is a pointer to an array of generic type `T`. The type `T` can be changed to `int`, `float`, etc., depending on the type of object to be created.

```
// vector.cpp: parameterized vector class
#include <iostream.h>
template <class T>
class vector
{
      T * v;         // changes to int *v, float *v, ..., etc.
      int size;      // size of vector v
   public:
      vector( int vector_size )
      {
         size = vector_size;
         v = new T[ vector_size ];  //e.g., v=new int[ size ],if T is int
      }
      ~vector()
      {
         delete v;
      }
      T & elem( int i )
      {
         if( i >= size )
            cout << endl << "Error: Out of Range";
         return v[i];
      }
      void show();
};
template <class T>
void vector<T>::show()
{
   for( int i = 0; i < size; i++ )
      cout << elem( i ) << ", ";
}
void main()
{
   int i;
   vector <int> int_vect( 5 );
   vector <float> float_vect( 4 );
   for( i = 0; i < 5; i++ )
      int_vect.elem( i ) = i + 1;
   for( i = 0; i < 4; i++ )
      float_vect.elem( i ) = float( i + 1.5 );
   cout << "Integer Vector: ";
   int_vect.show();
```

```
    cout << endl << "Floating Vector: ";
    float_vect.show();
}
```

Run

```
Integer Vector: 1, 2, 3, 4, 5,
Floating Vector: 1.5, 2.5, 3.5, 4.5,
```

Note that the class template specification is similar to an ordinary class specification except for the prefix `template <class T>` and the use of `T` in place of the data-type. This prefix informs the compiler that the class declaration following it is a template and uses `T` as a type name in the declaration. Thus, the class `vector` becomes a parameterized class with the type `T` as its parameter. The type `T` may be substituted by any data type including the user defined types.

In `main()`, the statements

```
vector <int> int_vect( 5 );
vector <float> float_vect( 4 );
```

create the vector objects `int_vect` and `float_vect` to hold vectors of type integer and floating point respectively. Once the objects of class template are created, their usage is same as the objects of non-template classes.

3.8 Exception Handling

An exceptional condition is an error situation that occurs during the normal flow of events and prevents the program from continuing correctly. C++ provides *exception handling mechanism* for handling error conditions that should not be ignored by a caller. Error condition such as division of a number by zero is difficult to predict; however, that can be handled by using exceptions.

C++ offers the following three constructs for handling exceptions:

- `try`
- `throw`
- `catch`

A block of code in which an exception can occur must be prefixed by the keyword `try`. This block of code is called *try-block*. It indicates that the program is prepared for testing the existence of exceptions. If an exception occurs, the program flow is interrupted; call to an exception handler is made if one exists, otherwise, `abort()` is invoked.

The exception handler is indicated by the `catch` keyword and it must be specified immediately after the *try-block*. The keyword `catch` can occur immediately after another `catch`. Each handler will only evaluate an exception that matches, or can be converted to, the type specified in its argument list. Every exception thrown by the program must be caught and processed by the exception handler. If the program fails to provide an exception handler for a thrown exception, the program will call the terminate function.

The mechanism suggests that error handling code must perform the following tasks.

- Detect the problem causing exception (Hit the exception)
- Inform that an error has occured (Throw the exception)
- Receive the error information (Catch the exception)
- Take corrective actions (Handle the exceptions)

The program `number.cpp` illustrates the mechanism of handling exceptions. It has the class `number` to store an integer number, the member function `read()` to read a number from the console and the member function `div()` to perform the division operation. It raises an exception if an attempt is made to divide a number by zero.

```
// number.cpp: Divide Exceptions, divide by zero exceptions
#include <iostream.h>
class number
{
  private:
     int num;
  public:
     void read()
     {
        cin >> num;
     }
     class DIVIDE {};          // abstract class used in exceptions
     int div( number num2 )
     {
        if( num2.num == 0 )  // check for zero divisor if yes
           throw DIVIDE();   // raise exception
        else
           return num / num2.num;  // compute and return the result
     }
};
int main()
{
   number num1, num2;
   int result;
   cout << "Enter Number 1: ";
   num1.read();
   cout << "Enter Number 2: ";
   num2.read();
   // statements must be enclosed in try block if exception is to be raised
   try
   {
      cout << "trying division operation...";
      result = num1.div( num2 );
      cout << "succeeded" << endl;
   }
   catch( number::DIVIDE )    // exception handler block
   {
      // actions taken in response to exception
      cout << "failed" << endl;
      cout << "Exception: Divide-By-Zero";
      return 1;
   }
   // no exceptions, display result
   cout << "num1/num2 = " << result;
   return 0;
}
```

Run1

```
Enter Number 1: 10
Enter Number 2: 2
trying division operation...succeeded
num1/num2 = 5
```

Run2

```
Enter Number 1: 10
Enter Number 2: 0
trying division operation...failed
Exception: Divide-By-Zero
```

In `main()`, the try-block

```
try
{
   result = num1.div( num2 );
}
```

invokes the member function `div()` to perform the division operation using the function defined in the `number` class. (See Figure 3.6.)

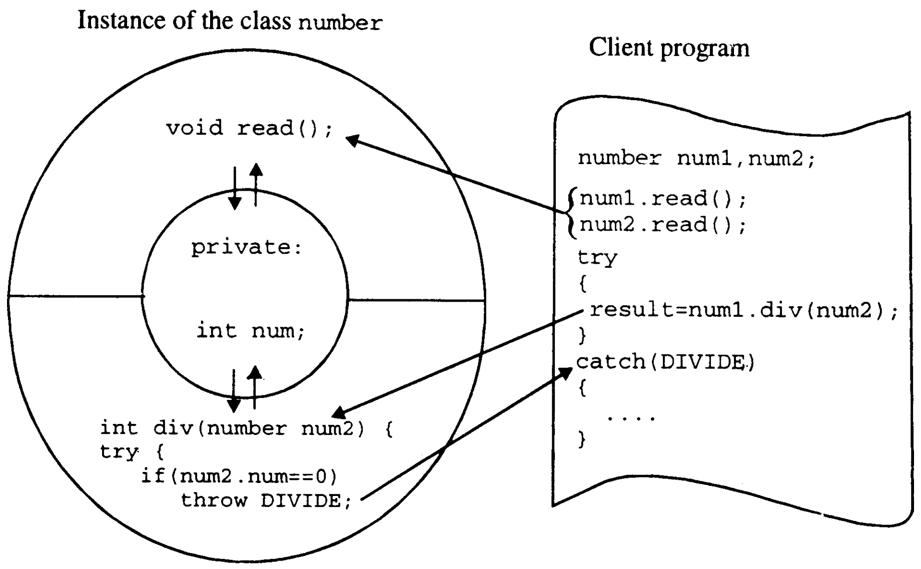


Figure 3.6: Exception handling in number class

If any attempt is made to divide by zero, the following statement in `div()` member function

```
if( num2.num == 0 )   // check for zero division if yes
   throw DIVIDE();    // raise exception
```

detects the same and raises the exception by passing a nameless object of the `DIVIDE` class. The following block of code in `main()` immediately after the try-block,

```
catch( number::DIVIDE )
{
   cout << "Exception: Divide-By-Zero";
   return 1;
}
```

will catch the exception raised due to a malfunction (divide-by-zero) in the preceding try-block and executes its (catch-block) body. When an exception is raised and if the exception matches with any of the catch's exception type, its catch-block will be executed; otherwise, the program terminates. The execution skips the catch-block and proceeds with the normal operations when no exception is raised.

3.9 Streams Computation

Stream is a name given to the flow of data and it acts as an interface between the program and the input/output devices. Streams provide a consistent interface irrespective of the device with which they operate (see Figure 3.7). For instance, the output operation can be performed either on the console or file; the interface for accessing these devices is the same as shown in the following statements:

```
cout << "Hello World";
outfile << "Hello World";
```

The first statement prints the message `Hello World` to a standard output device whereas the second statement prints the same in a file to which the variable `outfile` is the file handler.

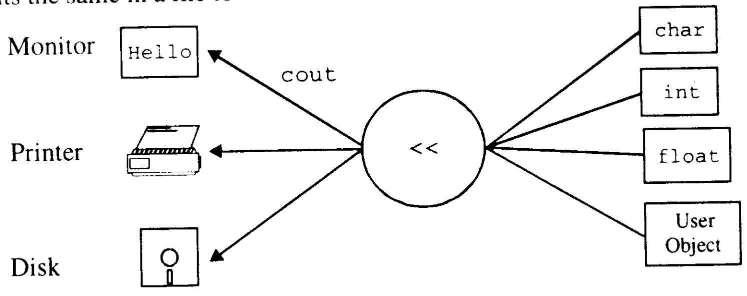


Figure 3.7: Consistent stream computation

Input-output operations in C++ are interpreted as a flow of stream of bytes. The program extracts bytes from the input stream when read operation is initiated and inserts bytes to the output stream when the output has to be performed.

C++ provides the following predefined stream objects (declared in iostream.h):

`cin`	Standard input (usually keyboard) corresponding to `stdin` in C.
`cout`	Standard output (usually screen) corresponding to `stdout` in C.
`cerr`	Standard error output (usually screen) corresponding to `stderr` in C.
`clog`	A fully-buffered version of `cerr` (no C equivalent).

The statement

```
cin >> m;
```

reads data from the console (keyboard) and stores it into the variable m. The statement

```
cout << "Hello World" << m;
```

prints the string message followed by the value stored in the variable m onto the console (monitor). The statement,

```
cerr << "Error: Hello World";
```

prints the string message onto the standard error device (usually monitor). The statement,

```
clog << "Log Errors";
```

prints the message to standard error device and displays when the buffer is flushed or \n (new line) character is encountered.

In C++, streams with operator overloading provide a mechanism for filtering. The standard stream operators << and >> do not know anything about the user-defined data types. They can be overloaded to operate on user-defined data items, which comprise operations on basic data items with standard stream operators. For example, consider the statements:

```
cout << counter1;

cin >> counter2;
```

The data-items counter1 and counter2, are the objects of the counter class (see friend.cpp program discussed above). The operators >> or << do not know anything about the objects counter1 and counter2. These are overloaded in the counter class as member functions, which process the attributes of counter objects as if they are basic data-items. Collectively, it appears as if the stream operators are operating on the objects of the class counter. This is possible due to overloading stream operators to operate on the user defined data types.

File Streams

A file is a unit of storage. The file handling technique of C does not support object oriented programming, hence C++ has come out with a new set of classes to deal with files.

As discussed earlier, the standard objects cin and cout have been used to deal with the standard input and the standard output. The objects cin and cout are declared in iostream.h header file. There are no such predefined objects for handling disk files. C++ supports the following classes for handling files:

- ifstream - for handling input files.
- ofstream - for handling output files.
- fstream - for handling files on which both input and output are done.

These classes are designed to manage the disk files and are declared in the fstream.h header file. To use file streams, include the following statement in the program:

```
#include <fstream.h>
```

The general pattern of accessing the data in a file is similar to the stdio.h functions. First, of course, the file has to be opened. In all the three classes, a file can be opened by giving a filename as the first parameter in the constructor itself. For example, the statement,

```
ifstream infile("test.txt");
```

will open the file test.txt for input operation.

The classes ifstream, ofstream, and fstream are derived from the classes istream, ostream, and iostream respectively to handle file streams and file input/output. The ifstream is meant for input files and ofstream for output files; the fstream is meant for both the input and output files.

File Input with `ifstream` Class

The class `ifstream` supports input operations. It contains the function `open()` with the default input mode. Inherits `get()`, `getline()`, `read()`, `seekg()`, and `tellg()` functions from `istream`. The program `infile.cpp` illustrates the use of `ifstream` class in file manipulation. It reads the contents of the file `sample.in` line by line and prints the same on the console.

```
// infile.cpp: reads all the names stored in file 'sample.in'
#include <fstream.h>
#include <process.h>
#include <iostream.h>
void main()
{
   char buff[ 80 ];
   ifstream infile;  // input file
   infile.open("sample.in");  // open file
   if( infile.fail())   // open fail
   {
     cout << "Error: sample.in non-existent";
     exit( 1 );
   }
   while( !infile.eof() )  // until end-of-file do processing
   {
     infile.getline(buff, 80);  // read complete line from file
     cout << buff << endl;
   }
   infile.close();
}
```

Run

```
Rajkumar, C-DAC, India
Bjarne Stroustrup, AT & T, USA
Smrithi, Hyderabad, India
Tejaswi, Bangalore, India
```

The input file `sample.in` contains the following information before the execution of the program:

```
Rajkumar, C-DAC, India
Bjarne Stroustrup, AT & T, USA
Smrithi, Hyderabad, India
Tejaswi, Bangalore, India
```

In `main()`, the statement

```
ifstream infile;                    // input file
```

creates the object `infile` and the statement

```
infile.open("sample.in");   // open file
```

opens the file `sample.in` in the input mode. The statement

```
if( infile.fail())          // open fail
```

checks for the status of file open operation. If file-open fails, it returns 1, otherwise 0. The statement

```
while( !infile.eof() )      // until end-of-file, do processing
```

repeats the file reading operation until the end-of-file. And the statement

```
        infile.getline(buff, 80);   // read complete line from file
```

reads a single line from the file or maximum of 80 characters from that line and proceeds to the next line. The statement,

```
        infile.close();
```

closes the file and thus preventing it from further manipulation.

File Output with `ofstream` Class

The class `ofstream` supports output operations. It contains the function `open()` with output mode as default. It inherits `put()`, `seekp()`, `tellp()`, and `write()` functions from `ostream`. The program `outfile.cpp` illustrates the use of the class `ofstream` in the file manipulation. It reads information entered through the keyboard and writes the same into the output file `sample.out`.

```
// outfile.cpp: writes all the input into the file 'sample.out'
#include <fstream.h>
#include <process.h>
#include <iostream.h>
#include <string.h>
void main()
{
   char buff[ 80 ];
   ofstream outfile; // output file
   outfile.open("sample.out");   // open in output mode
   if( outfile.fail())  // open fail
   {
      cout << "Error: sample.out unable to open";
      exit( 1 );
   }
   // loop until input = "end"
   while(1)
   {
      cin.getline(buff, 80);  // read a line from keyboard
      if( strcmp( buff, "end" ) == 0 )
         break;
      outfile << buff << endl;   // write to output file
   }
   outfile.close();
}
```

Run

```
OOP is good
C++ is OOP
C++ is good
end
```

Note: On execution, the file `sample.out` has the following:

```
OOP is good
C++ is OOP
C++ is good
```

In main(), the statement

```
    ofstream outfile;            // output file
```

creates the object outfile and the statement

```
    outfile.open("sample.out"); // open in output mode
```

opens the file sample.out in output mode. The statement

```
    if( outfile.fail())         // open fail
```

checks for the status of file open. If file open fails, it returns 1, otherwise 0. The statement

```
    outfile << buff << endl;    // write to output file
```

writes the buff contents and new-line character to the output file. The syntax of writing to the disk file resembles the writing to the console.

Guidelines

This chapter has given a glimpse on various prime features of C++. The fundamental construct of C++ i.e., *class* has been used to explain data encapsulation and abstraction features. More details on this can be found in chapters 10 and onwards. Other features discussed are inheritance, polymorphism, friend functions, virtual functions, class templates, exceptions handling, and streams computation.

Review Questions

3.1 State some reasons for C++ gaining popularity over other object-oriented programming languages.

3.2 Date consists of day, month, and year. Can this item be modeled as a class? What are the permissible operations this class needs to support ? Write a complete program having class declaration and the main() function to create its objects and manipulate them.

3.3 List the various object-oriented features supported by C++. Explain the constructs supported by C++ to implement them.

3.4 What is inheritance ? What are base and derived classes ? Give a suitable example for inheritance.

3.5 What are the different types of access specifiers supported by C++. Explain with a suitable example.

3.6 What is polymorphism ? Write a program to overload the + operator for manipulating objects of the Distance class.

3.7 What are friend functions ? Can they access members of a class directly ? Enhance the Date class such that it allows to read and display its objects using stream operators.

3.8 What are the differences between static binding and late binding ? Explain dynamic binding with a suitable example.

3.9 What are generic classes? Explain how they are useful. Write an interactive program having template-based Distance class. Create two objects: one of type integer and another of type floating-point.

3.10 What are exceptions ? What are the constructs supported by C++ to handle exceptions ?

3.11 What are streams ? Write an interactive program to copy a file to another file. Both source and destination files have to be processed as the objects of file-stream classes.

4

Data Types, Operators and Expressions

4.1 Introduction

Variables and constants are the fundamental elements of any programming language. Variables allow to name memory locations and use that name to access memory contents instead of accessing it through the physical address. Constants are those whose value never change during the execution of the program. Operators are used to specify the type of operation to be carried out on the variables and constants. Expressions combine the variables and constants to produce new values. The type of an object (variable/constant) determines the set of values it can represent and various operations that can be performed on it. When an expression has variables of different types, they need to be coerced (type converted) before their use. It can be either performed by the compiler implicitly, or by the user explicitly. C++ qualifiers allow promotion of any fundamental data type. The precedence and associativity of operators specify the order of evaluation of an expression to generate a valid output.

4.2 Character Set

The C++ character set consists of the upper and lower case alphabets, digits, special characters and white spaces. The alphabets and digits together constitute the alphanumeric set. The complete character set is shown in Table 4.1. The compiler ignores white spaces unless they are a part of a string constant. White spaces are used to separate words (and sometimes to increase the readability of a program), but cannot be embedded in the keywords and identifiers.

Alphabets:		
Uppercase: A B . . . Z		
Lowercase: a b . . . z		
Digits		
0 1 2 3 4 5 6 7 8 9		
Special Characters:		
, comma	< opening angle bracket	> closing angle bracket
. period	_ underscore	(left parenthesis
; semicolon	$ dollar sign	) right parenthesis
: colon	% percent sign	[left bracket
# number sign	? question mark	] right bracket
' apostrophe	& ampersand	{ left brace
" quotation mark	^ caret	} right brace
! exclamation mark	* asterisk	/ slash
| vertical bar	- minus sign	\ blackslash
~ tilde	+ plus sign	
White space characters:		
blank space	newline	carriage return
formfeed	horizontal tab	vertical tab

Table 4.1: C++ character set