invokes function template `max()` first time, the compiler creates `max()` which handles integer data. Future invocation such as,

```
i = max( j, k );            // i, j, and k are integers
```

accesses the function created at the first call since, the data type parameters `j` and `k` is the same as that of the first call. However, if `j` and `k` are other than integers, it creates a new function internally and makes a call to it.

Function and Function Template

Function templates are not suitable for handling all data types, and hence, it is necessary to override function templates by using normal functions for specific data types. When a statement such as

```
max( str1, str2 )
```

is executed, it will not produce the desired result. The above call compares memory addresses of strings instead of their contents. The logic for comparing strings is different from comparing integer or floating-point data type. It requires the function having the definition:

```
char * max(char * a, char * b)
{
   return( strcmp(a, b) > 0 ? a : b);
}
```

If the program has both the function and function template with the same name, first, the compiler selects the normal function, if it matches with the requested data type, otherwise, it creates a function using a function template. This is illustrated in the program `max2.cpp`.

```
// max2.cpp: maximum of standard and derived data type items
#include <iostream.h>
#include <string.h>
template <class T>
T max( T a, T b )
{
   if( a > b )
      return a;
   else
      return b;
}
// specifically for string data types
char * max( char *a, char *b )
{
   if( strcmp( a, b ) > 0 )
      return a;
   else
      return b;
}
void main()
{
   // max with character data types
   char ch, ch1, ch2;
   cout << "Enter two characters <ch1, ch2>: ";
   cin >> ch1 >> ch2;
   ch = max( ch1, ch2 );
   cout << "max( ch1, ch2 ): " << ch << endl;
```

```
    // max with integer data types
    int a, b, c;
    cout << "Enter two integers <a, b>: ";
    cin >> a >> b;
    c = max( a, b );
    cout << "max( a, b): " << c << endl;
    // max with string data types
    char str1[20], str2[20];
    cout << "Enter two strings <str1, str2>: ";
    cin >> str1 >> str2;
    cout << "max( str1, str2 ): " << max( str1, str2 );
}
```

Run

```
Enter two characters <ch1, ch2>: A Z
max( ch1, ch2 ): Z
Enter two integers <a, b>: 5 6
max( a, b): 6
Enter two strings <str1, str2>: Tejaswi Rajkumar
max( str1, str2 ): Tejaswi
```

In `main()`, the statement

```
cout << "max( str1, str2 ): " << max( str1, str2 );
```

has the expression,

```
max( str1, str2 )
```

The compiler selects the user-defined normal function instead of creating a new function, since the function call is matching with the user-defined function.

Bubble Sort Function Template

Sorting is the most commonly used operation particularly in data processing applications. These applications require function to sort data elements of different data types. Such functions can be declared as function template and can be used to sort data items of any type. The program `bsort.cpp` illustrates the declaration of function template for bubble sort and its use on integer and floating point data types.

```
// bsort.cpp: template functions for bubble-sort
#include <iostream.h>
enum boolean { false, true };
template <class T>
void swap( T & x, T & y ) // by reference
{
   T t;    // template type temporary variable used in swapping
   t = x;
   x = y;
   y = t;
}
template< class T >
void BubbleSort( T & SortData, int Size )
{
   boolean swapped = true;
```

```
    for( int i = 0; ( i < Size - 1 ) && swapped; i++ )
    {
        swapped = false;
        for( int j = 0; j < ( Size - 1 ) - i; j++ )
            if( SortData[ j ] > SortData[ j + 1 ] )
            {
                swapped = true;
                swap( SortData[ j ], SortData[ j + 1 ] );
            }
    }
}
void main( void )
{
    int IntNums[25];
    float FloatNums[25];
    int i, size;
    cout << "Program to sort elements..." << endl;
    // Integer numbers sorting
    cout << "Enter the size of the integer vector <max-25>: ";
    cin >> size;
    cout << "Enter the elements of the integer vector..." << endl;
    for( i = 0; i < size; i++ )
        cin >> IntNums[i];
    BubbleSort( IntNums, size );
    cout << "Sorted Vector:" << endl;
    for( i = 0; i < size; i++ )
        cout << IntNums[i] << "  ";
    // Floating point numbers sorting
    cout << "\nEnter the size of the float vector <max-25>: ";
    cin >> size;
    cout << "Enter the elements of the float vector..." << endl;
    for( i = 0; i < size; i++ )
        cin >> FloatNums[i];
    BubbleSort( FloatNums, size );
    cout << "Sorted Vector:" << endl;
    for( i = 0; i < size; i++ )
        cout << FloatNums[i] << "  ";
}
```

Run

```
Program to sort elements...
Enter the size of the integer vector <max-25>: 4
Enter the elements of the integer vector...
8
4
1
6
Sorted Vector:
1  4  6  8
Enter the size of the float vector <max-25>: 3
```

```
Enter the elements of the float vector...
8.5
3.2
8.9
Sorted Vector:
3.2  8.5  8.9
```

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

```
BubbleSort( IntNums, size );
```

it creates the bubble sort function internally for sorting integer numbers; the parameter `IntNums` is of type integer. Similarly, when the compiler encounters the statement

```
BubbleSort( FloatNums, size );
```

it creates bubble sort function internally for sorting floating point numbers. The same template function can be used to sort any other data types. Note that the compiler creates a function internally for a particular data type only once and if there are more requests with the same data type, the compiler accesses the old internally created function.

Usage of Template Arguments

Every template-argument specified in the template-argument-list *must* be used as a generic data type for the definition of the formal parameters. If any of the generic data type is not used in the definition of formal parameters, such function templates are treated as invalid templates. The use of partial number of generic data types in a function defining formal parameters is also treated as an error. All the formal parameters need not be of generic type. The following sections show some function templates which are invalid declarations:

1. *No-argument template function*

```
template < class T >
T pop( void )          // error: T is not used as an argument
{
   return *--Stack;
}
```

2. *Template-type argument unused*

```
template < class T >
void test( int x )     // error: T is not used as an argument
{
   T temp;
   // .. test stuff
}
```

3. *Usage of Partial number of template arguments*

```
template< class T, class U >
void insert( T & x )  // error: U is not used in the argument
{
   U lPtr;
   // .. test stuff
}
```

The template argument `U` is not used in argument types, and hence, the compiler reports an error.

16.3 Overloaded Function Templates

The functions templates can also be overloaded with multiple declarations. It may be overloaded either by (other) functions of its name or by (other) template functions of the same name. Similar to overloading of normal functions, overloaded functions must differ either in terms of number of parameters or their type. The program `tprint.cpp` illustrates the overloading of function templates:

```
// tprint.cpp: overloaded template functions
#include <iostream.h>
template <class T>
void print( T data )    // single template argument
{
   cout << data << endl;
}
template <class T>
void print( T data, int nTimes) // template and standard argument
{
   for( int i = 0; i < nTimes; i++ )
      cout << data << endl;
}
void main()
{
   print( 1 );
   print( 1.5 );
   print( 520, 2 );
   print( "OOP is Great", 3 );
}
```

Run

```
1
1.5
520
520
OOP is Great
OOP is Great
OOP is Great
```

In the above program, the templates

```
void print( T data )    // single template argument
void print( T data, int nTimes)   // template and standard argument
```

overload the function template `print()`, but each one of these functions is distinguishable by the number of arguments and the type of the arguments. In `main()`, the statements

```
print( 1 );
print( 1.5 );
```

access the one-argument function template whereas, the statements

```
print( 520, 2 );
print( "OOP is Great", 3 );
```

access the two argument function template. Note that in these statements, the required function is selected based on the number of arguments supplied at the point of call.

The compiler adopts the following rules for selecting a suitable template when the program has overloaded function templates.

[1] Look for an exact match on functions; if found, call it.

[2] Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.

[3] Try ordinary overloading resolution for the functions; if found, call it.

If no match is found in all the three alternatives, then that call is treated as an error. In each case if there is more than one alternative in the first step that finds a match, the call is ambiguous and is an error.

A match on a template (step [2]) implies that a specific template function with arguments that exactly matches the types of the arguments will be generated. In this case, not even trivial type-conversion is applied while matching a call to a function template.

16.4 Nesting of Function Calls

Recursively designed algorithms will have nested calls to themselves. Their implementation in the form of function templates will also have recursive calls (calls to itself). The binary search can be implemented by using recursion. It searches for an item in a list of ordered data by applying the ***divide and conquer*** strategy. The program `bsearch.cpp` illustrates the template based implementation of recursive binary search algorithm.

```
// bsearch.cpp: binary search function template
#include <iostream.h>
enum boolean { false, true };
// recursive binary search
template <class T>
int RecBinSearch( T Data[], T SrchElem, int low, int high )
{
   if( low > high )
      return -1;
   int mid = ( low + high ) / 2;
   if( SrchElem < Data[mid] )
      return RecBinSearch( Data, SrchElem, low, mid - 1 );
   else
      if( SrchElem > Data[mid] )
         return RecBinSearch( Data, SrchElem, mid + 1, high );
   return mid;
}
void main( void )
{
   int elem, size, num[25], index;
   cout<< "Program to search integer elements..." << endl;
   cout << "How many elements ? ";
   cin >> size;
   cout<<"Enter the elements in ascending order for binary search..."<<endl;
   for( int i = 0;i < size; i++ )
      cin >> num[i];
   cout << "Enter the element to be searched: ";
```

```
    cin >> elem;
    if( ( index = RecBinSearch( num, elem, 0, size ) ) == -1 )
       cout << "Element " << elem << " not found" << endl;
    else
       cout << "Element " << elem << " found at position " << index;
}
```

Run

```
Program to search integer elements...
How many elements ? 4
Enter the elements in ascending order for binary search...
1
4
6
8
Enter the element to be searched: 6
Element 6 found at position 2
```

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

```
RecBinSearch( num, elem, 0, size )
```

it creates the search function internally. The function `RecBinSearch()` has recursive calls to itself. In this case, the compiler will not create a new function instead, it uses the internally created function.

16.5 Multiple Arguments Function Template

So far, all the function templates dealt with a single generic argument. Declaration of a function template for functions having multiple parameters of different types requires multiple generic arguments. The program `multiple.cpp` illustrates the need for multiple template arguments.

```
// multiple.cpp use of multiple template arguments
struct A
{
   int x;
   int y;
};
struct B
{
   int x;
   double y;
};
template < class T >
void Assign_A( T a, T b, A & S1 )
{
   S1.x = a;
   S1.y = b;
}
template < class T >
void Assign_B( T a, T b, B & S2 )
{
   S2.x = a;
```

```
   S2.y = b;
}
void main( void )
{
   A S1;
   B S2;
   Assign_A( 3, 4, S1 );
   Assign_B( 3, 3.1415, S2 );//Error: no match for Assign_B(int,double,B)'
}
```

In `main()`, the statement

```
Assign_B( 3, 3.1415, S2 );
```

leads to compilation errors, since the above program is neither having the normal function nor function template matching with its parameters data types. Both the templates expect the first two parameters to be of the same data-type and none of them matches the above call. The solution to the problem encountered in the above program is to declare the second template function in the above program as follows:

```
template < class T, class U >
void Assign_B( T a, U b, B & S2 )
{
   S2.x = a;
   S2.y = b;
}
```

The declaration of the function template is the same, except that it has an extra argument in the template-argument-list, i.e., class `U`. This declaration informs the compiler that the template function `Assign_B()` with two arguments should be instantiated. The compiler calls the appropriate instantiation. Any number of generic data types can be declared, provided all these generic data types are used in declaring formal parameters. The function `Assign_A` can also be declared as follows:

```
template< class T, class U >
void Assign_A( T a, U b, A & S2 )
{
   S1.x = a;
   S1.y = b;
}
```

Since the dummy arguments `T` and `U` are same in the function call `Assign_A`, it would be better to define the function template with a single dummy argument rather than two dummy arguments.

All template arguments for a function template must be of template type-arguments, otherwise, it leads to an error. For instance, the following declaration,

```
template< class T, unsigned SIZE >
void BubbleSort( T & Data, unsigned SIZE )
{
   //....
   //....
}
```

is not allowed. However, such declarations are allowed with class templates.

16.6 User Defined Template Arguments

In addition to primitive data-types, user defined types can be passed to function templates. Its declaration is same as the function template processing standard data types as illustrated in the program `student.cpp`.

```
// student.cpp: student record and template with user defined data types
#include <iostream.h>
struct stuRec
{
   char name[30];
   int age;
   char collegeCode;
};
template <class T>
void Display( T& t )
{
   cout << t << endl;
}
ostream& operator << ( ostream & out, stuRec & s )
{
   out << "Name: " << s.name << endl;
   out << "Age : " << s.age << endl;
   out << "College Code: " << s.collegeCode << endl;
   return out;
}
void main( void )
{
   stuRec s1;
   cout << "Fnter student record details..." << endl;
   cout << "Name: "; cin >> s1.name;
   cout << "Age : "; cin >> s1.age;
   cout << "College Code: "; cin >> s1.collegeCode;
   cout << "The student record:" << endl;
   cout << "Name: "; Display( s1.name );
   cout << "Age : "; Display( s1.age );
   cout << "College Code: ";
   Display( s1.collegeCode ); // it in turn calls operator << defined above
   cout << "The student record:" << endl;
   Display( s1 );
}
```

Run

```
Enter student record details...
Name: Chinamma
Age : 18
College Code: A
The student record:
Name: Chinamma
Age : 18
```

```
College Code: A
The student record:
Name: Chinamma
Age : 18
College Code: A
```

In `main()`, the statement

```
Display( s1 );
```

accesses the function template; the statement

```
cout << t << endl;
```

in `Display()` invokes the overloaded operator function,

```
ostream& operator << ( ostream & out, stuRec & s )
```

In the `cout` statement, when the compiler encounters the user defined data item, it searches for the overloaded stream operator function and makes a call to it.

16.7 Class Templates

Similar to functions, classes can also be declared to operate on different data types. Such classes are called *class templates*. A class template specifies how individual classes can be constructed similar to normal class specification. These classes model a generic class which support similar operations for different data types. A generic stack class can be created, which can be used for storing data of type integer, real, double, etc. Consider an example of a stack (modeling last-in-first-out data structure) to illustrate the need and benefits of class templates. The class declaration for stacks of type character, integer, and double would be as follows:

```
class CharStack
{
   char array[25];          // declare a stack of 25 characters
   unsigned int top;
   public:
      CharStack();
      void Push( const char & element );
      char Pop( void );
      unsigned int GetSize( void ) const;
};
class IntStack
{
   int array[25];           // declare a stack of 25 integers
   unsigned int top;
   public:
      IntStack();
      void Push( const int & element );
      int Pop( void );
      unsigned int GetSize( void ) const;
};
class DbleStack
{
   double array[25];        // declare a stack of 25 double
   unsigned int top;
```

```
    public:
       DbleStack();
       void Push( const double & element );
       double Pop( void );
       unsigned int GetSize( void ) const;
};
```

As seen in the above three declarations, a separate stack class is required for each and every data type. Template declaration enables substitution of code for all the three declarations of stacks with a single template class as follows:

```
template < class T >
class DataStack
{
   T array[25];      // declare a stack of 25 elements of data type T
   unsigned int top;
   public:
      DataStack();
      void Push( const T & element );
      T Pop( void );
      unsigned int GetSize( void ) const;
};
```

The syntax of declaring class templates and defining objects using the same is shown in Figure 16.2. The definition of a class template implies defining template data and member functions.

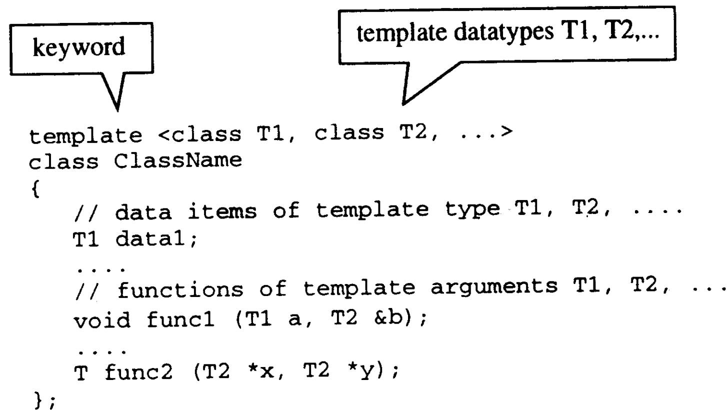


Figure 16.2: Syntax of class template declaration

The prefix `template <class T>` specifies that a template is being declared, and that a *type-name* T will be used in the declaration. In other words, `DataStack` is a parameterized class with T as its generic data type.

Any call to the template functions and classes, needs to be associated with a data type or a class. The compiler then instantiates a copy of the template function or template class for the data type specified. The syntax for creating objects using the class templates is shown in Figure 16.3.

A statement to create an object of type stack that can hold integers is as follows:

```
DataStack <int> stack_int; // stack of integers
```

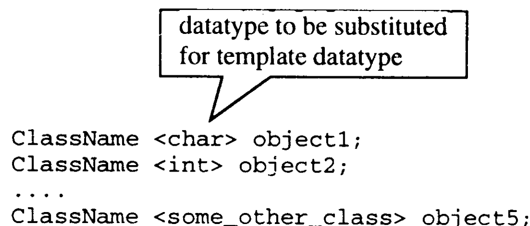


Figure 16.3: Syntax for class template instantiation

Similarly, objects those hold characters, floats, and doubles can be created by the following statements:

```
DataStack <char> stack_char;       // stack of characters
DataStack <float> stack_float;     // stack of floats
DataStack <double> stack_double;   // stack of doubles
```

However, the usage of these stack objects is similar to those of normal objects. For instance, to push the integer value 10 into the `stack_int` object, the following statement is used:

```
stack_int.push( 10 );
```

Template Arguments

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

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

The value supplied for a non template type argument must be a constant expression; its value must be known at compile time. When the objects of the class template are created using a 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];
};
```

Member Function Templates

A member function of a template class is implicitly treated as a template function and it can have

template arguments which are the same as its class template arguments. For instance, the class template `DataStack` has the member function,

```
void Push( const T &element );
```

The parameter element is of type template-argument. Its syntax when defined outside is as follows:

```
template <class T>
void DataStack <T>::Push( const T &element );
```

The syntax for declaring member functions of a template class outside its body is shown in Figure 16.4.

```
template <class T1, ...>
class BaseClass
{
   // template type data and functions
   void func1(T1 a);
};
template <class T1, ...>
void ClassName <T1,...>::func1(T1 a)
{
   // function template body
};
```

Figure 16.4: Syntax for declaring member function of class template outside its body

The program `vector.cpp` illustrates the declaration of the `vector` class and its usage in defining its objects. It has a data member which is a pointer to an array of type T. The type T can be `int`, `float`, etc., depending on the type of the object created.

```
// vector.cpp: parametrized 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 ];  // v = new int[ size ], ...
      }
      ~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 very much similar to the ordinary class specification except for the prefix,

```
template <class T>
```

and the use of `T` in the place of 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. The type `T` may be substituted by any data type including the user defined types. In `main()`, the statement,

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

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

Class Template with Multiple Arguments

The template class is instantiated by specifying predefined data type or user defined data classes. The data type is specified in angular braces <>. The syntax for instantiating class template is as follows:

```
TemplateClassName < type > instance;
TemplateClassName < type1, type2 > instance( arguments );
```

The instantiation specifies the objects of specified data type. If a different data type is to be specified, a new declaration statement must be used.

The declaration of template classes with multiple arguments is similar to the function template with multiple arguments. However, the arguments need not be of template type. These may include character strings, addresses of objects and functions with external linkage, static class members, and constant expressions. Consider the following declaration:

```
template < class T, unsigned SIZE >
class StackN
{
  protected:
      T Array[SIZE];
      unsigned int top;
  public:
      Stack20( ) { top = 0; }
      void Push( const T & elem ) { Array[ top++ ] = elem; }
      T Pop( void ) { return Array[ --top ]; }
      int GetSize( void ) const { return top+1; }
      T & GetTop( void ) { return Array[top]; }
};
```

The declaration of the class template `StackN` is preceded by,

```
template < class T, unsigned SIZE >
```

as before, except that it has two arguments. The second argument is an (typed) unsigned argument. Making `SIZE` an argument of the template class `StackN` rather than to its objects, infers that the sizes of class `StackN` is known at compile time so that class `StackN` can be fully declared at compile time. The class template `StackN` with a variable stack size can be instantiated by specifying the size in the argument list. This makes a template, such as `StackN`, useful for implementing general purpose data structure. The above declarations provide the user freedom to define many instances of the class `StackN`, each operating on different data-types and of variable size. The following statements define objects of the class template `StackN` for storing integers and characters respectively.

```
StackN < int, 20> Intstk;
StackN < char, 50 > Chrstk;
```

A known type argument in the template class (second argument in the above case) must be a constant expression (evaluated at the compile time) of the appropriate type.

The list allows insertion operation at the front and deletion operation at the end of a list. The list class can have any number of template data elements, as shown in the following declaration.

```
template< class R, class S, class T >
class SnglList
{
  private:
    R data_1;
    S data_2;
    T data_3;
  public:
    SnglList< R, S, T > *next;
    SnglList( void ) { next = NULL; }
    .....
    friend ostream & operator<<( ostream &, SnglList< R, S, T > & );
    friend istream & operator>>( istream &, SnglList< R, S, T > & );
};
```

The objects of class templates having multiple arguments can be created as follows:

```
SnglList <int, float, double> node;
SnglList < int, unsigned, double > *Root, *End;
```

16.8 Inheritance of Class Template

A combination of templates and inheritance can be used in developing hierarchal data structures such as container classes. A base class in a hierarchy represents a commonality of methods and properties. Use of templates with respect to inheritance involves the following:

- Derive a class template from a base class, which is a template class.
- Derive a class template from the base class, which is a template class, add more template members in the derived class.
- Derive a class from a base class which is not a template, and add template members to that class.
- Derive a class from a base class which is a template class and restrict the template feature, so that the derived class and its derivatives do not have the template feature.

The template features provided in the base classes, can be restricted by specifying the type, when the class is derived. All the arguments in the template argument list of the base class have to be replaced by predefined types. In such a case, the derived class does not inherit the template feature, but is just a class of *specified data type* stated at the point of inheritance declaration. The syntax for declaring derived classes from template-based base classes is shown in Figure 16.5.

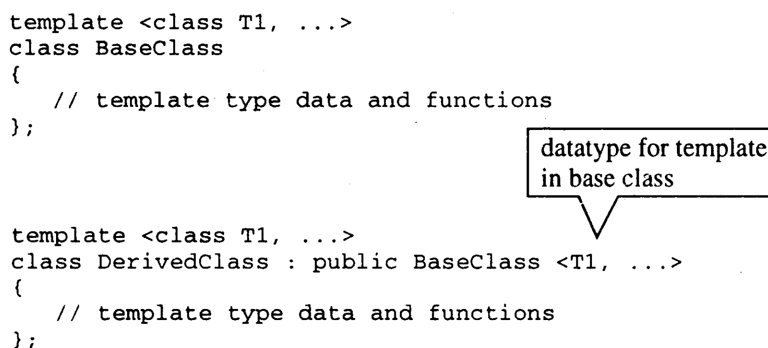


Figure 16.5: Syntax for inheriting template base class

The class deriving a template type base class can be a normal class or a class-template. If a new derived class is a normal class, the data-type of template arguments to the base class must be specified at the point of derivation. Otherwise, template arguments type specified at the point of instantiation of a class template can also be passed.

Consider an example of declaring the template class `Vector`. It inherits all the properties from the base template class `sVector`. The derived template class `Vector` is still a static vector containing twenty elements. Member functions that perform insert, delete and search are added to the derived class. The member functions have the prefix `<template class T>`, since the derived class operates on the undeclared type `T`. The specification of a new template class created by inheriting another template-based base class is given below:

```
template< class T >
class Vector : public sVector< T >
{
   ...
   read();
```

```
    ...
};
```

The member functions defined with its class body have the same syntax as members of non-template-type classes. However, member function defined outside the body of a class, for instance, has the following specification:

```
template <class T>
void Vector<T>::read()
{
  ...// body of the read()
}
```

Note that, the member functions of a class-template are treated as function-template type members. The class `Vector` can be instantiated as follows:

```
Vector <int> v1;
```

In this case, the `int` specified in angular bracket is first assigned to generic data type in the `Vector` class and then the same is also passed to its base class.

A derived class of a template based base class is not necessarily template derived class. That is, the non-template-based derived classes can also be created from the template-based base classes. In this case, the undefined template argument T has to be specified during derivation, for instance, as follows:

```
class Vector : public sVector< int >
{
    ...
};
```

It creates a new class called `Vector` from the template-based base class `sVector`. The `int` is passed as template argument type to the base class.

The program `union.cpp` illustrates the mechanism of extending the class template `Bag` by using the feature of inheritance. In this case, a new class template `Set` is derived from the existing class template `Bag` without any modifications. A derived class template `Set` inherits all the properties of the class template `Bag` and extends itself by adding some more features of its own to support set assignment and union operation.

```
// union.cpp: Union of sets. Set class by inheritance of Bag class
#include <iostream.h>
enum boolean { FALSE, TRUE };
const int MAX_ITEMS = 25;  // Maximum number of items that the bag can hold

template <class T>
class Bag
{
  protected:                // Note: not private
    T contents[MAX_ITEMS];  // bag memory area
    int ItemCount;          // Number of items present in the bag
  public:
    Bag()        // no-argument constructor
    {
      ItemCount = 0;     // When you purchase a bag, it will be empty
    }
```

```
    void put( T item )    // puts item into bag
    {
      contents[ ItemCount++ ] = item;  // item into bag, counter update
    }
    boolean IsEmpty()     // 1, if bag is empty, 0, otherwise
    {
      return ItemCount == 0 ? TRUE : FALSE;
    }
    boolean IsFull()      // 1, if bag is full, 0, otherwise
    {
      return ItemCount == MAX_ITEMS ? TRUE : FALSE;
    }
    boolean IsExist( T item );
    void show();
};
// returns 1, if item is in bag, 0, otherwise
template <class T>
boolean Bag<T>::IsExist( T item )
{
   for( int i = 0; i < ItemCount; i++ )
      if( contents[i] == item )
         return TRUE;
   return FALSE;
}
// display contents of a bag
template <class T>
void Bag<T>::show()
{
   for( int i = 0; i < ItemCount; i++ )
      cout << contents[i] << " ";
   cout << endl;
}
template <class S>
class Set: public Bag <S>
{
   public:
      void add( S element )
      {
         if( !IsExist( element ) && !IsFull() )
            put( element );
      }
      void read();
      void operator = (Set s1);
      friend Set operator + ( Set s1, Set s2 );
};
template <class S>
void Set<S>::read()
{
   S element;
   while( TRUE )
   {
```

```
      cout << "Enter Set Element <0- end>: ";
      cin >> element;
      if( element == 0 )
         break;
      add( element );
   }
}
template <class S>
void Set<S>::operator = ( Set <S> s2 )
{
   for( int i = 0; i < s2.ItemCount; i++ )
      contents[i] = s2.contents[i];
   ItemCount = s2.ItemCount;
}
template <class S>
Set<S> operator + ( Set <S> s1, Set <S> s2 )
{
   Set <S> temp;
   temp = s1;    // copy all elements of set s1 to temp
   // copy those elements of set s2 into temp, those not exist in set s1
   for( int i = 0; i < s2.ItemCount; i++ )
   {
      if( !s1.IsExist( s2.contents[i] ) ) // if element of s2 is not in s1
         temp.add( s2.contents[i] );      // copy the unique element
   }
   return( temp );
}
void main()
{
   Set <int> s1;
   Set <int> s2;
   Set <int> s3;
   cout << "Enter Set 1 elements .." << endl;
   s1.read();
   cout << "Enter Set 2 elements .." << endl;
   s2.read();
   s3 = s1 + s2;
   cout << endl << "Union of s1 and s2 : ";
   s3.show();     // uses Bag::show() base class
}
```

Run

```
Enter Set 1 elements ..
Enter Set Element <0- end>: 1
Enter Set Element <0- end>: 2
Enter Set Element <0- end>: 3
Enter Set Element <0- end>: 4
Enter Set Element <0- end>: 0
Enter Set 2 elements ..
Enter Set Element <0- end>: 2
```

```
Enter Set Element <0- end>: 4
Enter Set Element <0- end>: 5
Enter Set Element <0- end>: 6
Enter Set Element <0- end>: 0
Union of s1 and s2 : 1 2 3 4 5 6
```

In the above program, the template class `Set` has its own features to perform set union by using the member functions of the class `Bag`. The statement

```
template <class S>
class Set: public Bag <S>
```

derives the new template class `Set` known as derived class from the base class `Bag`. The base class `Bag` is *publicly inherited* by the derived class `Set`. Hence, the members of `Bag` class, which are *protected* remain *protected* and *public* remain *public*, in the derived class `Set`. The `Set` class can treat all the members of the `Bag` class as they are of its own. The derived class `Set` refers to the data and member functions of the base class `Bag`, while the base class `Bag` has no access to the derived class `Set`.

16.9 Class Template Containership

The usage of delegation (containership) with templates allows to build powerful programming components (data structures). It refers to having an object of one class contained in another class as a data member. The container class (i.e., a class that holds objects of some other type) is of considerable importance when implementing data structures. Inheritance supports the *is-a* relationship whereas containership supports the *has-a* relationship. The program `tree.cpp` illustrates the use of containership in building an unbalanced binary tree. It has two classes `TreeNode` and `BinaryTree`. The first class represents the node structure of a binary tree where as the second class represents the set of operations which can be performed on a tree. The class `TreeNode` has two pointers to objects of its own which serve as the pointers to child nodes. The class `BinaryTree` has a pointer to the root node of the tree, which is an instance of the class `TreeNode` and thus delegating node handling issues to the `TreeNode` class.

```
// tree.cpp: Binary Tree Operations (create, print, traverse, and search)
#include <iostream.h>
#include <stdio.h>
template <class T>
class TreeNode
{
  protected:
     T data;                  /* data to be stored in a tree */
     TreeNode <T> *left;      /* pointer to a left sub tree */
     TreeNode <T> *right;     /* pointer to a right sub tree */
  public:
     TreeNode( const T& dataIn)
     {
        data = dataIn;
        left = right = NULL;
     }
```

```
        TreeNode( const T& dataIn, TreeNode <T> *l, TreeNode <T> *r )
        {
            data = dataIn;
            left = l;
            right = r;
        }
        friend class BinaryTree <T>;
};
template <class T>
class BinaryTree
{
    protected:
        TreeNode<T> *root;
        TreeNode<T> *InsertNode( TreeNode <T> *root, T data );
    public:
        BinaryTree()
        {
            root = NULL;
        }
        void PrintTreeTriangle( TreeNode <T> *tree, int level );
        void PrintTreeDiagonal( TreeNode <T> *tree, int level );
        void PreOrderTraverse( TreeNode <T> *tree );
        void InOrderTraverse( TreeNode <T> *tree );
        void PostOrderTraverse( TreeNode <T> *tree );
        TreeNode <T> * SearchTree( TreeNode <T> *tree, T data );
        void PreOrder()
        {
            PreOrderTraverse( root );
        }
        void InOrder()
        {
            InOrderTraverse( root );
        }
        void PostOrder()
        {
            PostOrderTraverse( root );
        }
        void PrintTree( int disptype )
        {
            if( disptype == 1 )
                PrintTreeTriangle( root, 1);
            else
                PrintTreeDiagonal( root, 1);
        }
        void Insert( T data )
        {
            root = InsertNode( root, data );
        }
```

```
      TreeNode <T> * Search( T data )
      {
         return SearchTree( root, data );
      }
};
// Insert 'data' into tree
template <class T>
TreeNode<T> * BinaryTree<T>::InsertNode( TreeNode <T> *tree, T data )
{
   /* Is Tree NULL */
   if( !tree )
   {
      tree =  new TreeNode<T>( data, NULL, NULL );
      return( tree );
   }
   /* Is data less than the parent element */
   if( data < tree->data )
      tree->left = InsertNode( tree->left, data );
   else
      /* Is data greater than the parent element */
      if( data > tree->data )
         tree->right = InsertNode( tree->right, data );
   /* data already exists */
   return( tree );
}
// PreOrder Traversal
template <class T>
void BinaryTree<T>::PreOrderTraverse( TreeNode <T> *tree )
{
   if( tree )
   {
      cout << tree->data << " "; // Process node
      PreOrderTraverse( tree->left );
      PreOrderTraverse( tree->right );
   }
}
// In Order Traversal
template <class T>
void BinaryTree<T>::InOrderTraverse( TreeNode <T> *tree )
{
   if( tree )
   {
      PostOrderTraverse( tree->left );
      cout << tree->data << " "; // Process node
      PostOrderTraverse( tree->right );
   }
}
```

```
// Post Order Traversal
template <class T>
void BinaryTree<T>::PostOrderTraverse( TreeNode <T> *tree )
{
   if( tree )
   {
      PostOrderTraverse( tree->left );
      PostOrderTraverse( tree->right );
      cout << tree->data << " "; // Process node
   }
}
// Tree Printing in Triangle Form
template <class T>
void BinaryTree<T>::PrintTreeTriangle( TreeNode <T> *tree, int level )
{
   if( tree )
   {
      PrintTreeTriangle( tree->right, level+1 );
      cout << "\n";
      for( int i = 0; i < level; i++ )
         cout << "     ";
      cout << tree->data;
      PrintTreeTriangle( tree->left, level+1 );
   }
}
// Tree Printing in Diagonal Form
template <class T>
void BinaryTree<T>::PrintTreeDiagonal( TreeNode <T> *tree, int level )
{
   if( tree )
   {
      cout << "\n";
      for( int i = 0; i < level; i++ )
         cout <<  "     ";
      cout << tree->data;
      PrintTreeDiagonal( tree->left, level+1 );
      PrintTreeDiagonal( tree->right, level+1 );
   }
}
// search for data item in the tree
template <class T>
TreeNode <T> * BinaryTree<T>::SearchTree( TreeNode <T> *tree, T data )
{
   while( tree )
   {
      /* Is data less than the parent element */
      if( data < tree->data )
         tree = tree->left;
      else
```

```
        /* Is data greater than the parent element */
        if( data > tree->data )
           tree = tree->right;
        else
           return( tree );
   }
   return( NULL );
}
void main()
{
   float data, disptype;
   BinaryTree <float> btree;    // tree's root node
   cout << "This Program Demonstrates the Binary Tree Operations" << endl;
   cout << "Tree Display Style: [1] - Triangular [2] - Diagonal form: ";
   cin >> disptype;
   cout << "Tree creation process..." << endl;
   while( 1 )
   {
      cout << "Enter node number to be inserted <0-END>: ";
      cin >> data;
      if( data == 0 )
         break;
      btree.Insert( data );
      cout << "Binary Tree is...";
      btree.PrintTree( disptype );
      cout << "\n Pre-Order Traversal: ";
      btree.PreOrder();
      cout << "\n  In-Order Traversal: ";
      btree.InOrder();
      cout << "\nPost-Order Traversal: ";
      btree.PostOrder();
      cout << endl;
   }
   cout << "Tree search process..." << endl;
   while( 1 )
   {
      cout << "Enter node number to be searched <0-END>: ";
      cin >> data;
      if( data == 0 )
         break;
      if( btree.Search( data ) )
         cout << "Found data in the Tree" << endl;
      else
         cout << "Not found data in the Tree" << endl;
   }
}
```

Run

```
This Program Demonstrates the Binary Tree Operations
Tree Display Style: [1] - Triangular [2] - Diagonal form: 1
```

```
Tree creation process...
Enter node number to be inserted <0-END>: 5
Binary Tree is...
     5
 Pre-Order Traversal: 5
  In-Order Traversal: 5
Post-Order Traversal: 5
Enter node number to be inserted <0-END>: 3
Binary Tree is...
     5
          3
 Pre-Order Traversal: 5 3
  In-Order Traversal: 3 5
Post-Order Traversal: 3 5
Enter node number to be inserted <0-END>: 8
Binary Tree is...
          8
     5
          3
 Pre-Order Traversal: 5 3 8
  In-Order Traversal: 3 5 8
Post-Order Traversal: 3 8 5
Enter node number to be inserted <0-END>: 2
Binary Tree is...
          8
     5
          3
               2
 Pre-Order Traversal: 5 3 2 8
  In-Order Traversal: 2 3 5 8
Post-Order Traversal: 2 3 8 5
Enter node number to be inserted <0-END>: 9
Binary Tree is...
               9
          8
     5
          3
               2
 Pre-Order Traversal: 5 3 2 8 9
  In-Order Traversal: 2 3 5 9 8
Post-Order Traversal: 2 3 9 8 5
Enter node number to be inserted <0-END>: 0
Tree search process...
Enter node number to be searched <0-END>: 8
Found data in the Tree
Enter node number to be searched <0-END>: 1
Not found data in the Tree
Enter node number to be searched <0-END>: 0
```

16.10 Class Template with Overloaded Operators

The class template can also be declared for a class having operator overloaded member functions . The syntax for declaring operator overloaded functions is the same as class template members and overloaded functions. The class template with operator overloading will allow to truly extend the language and at the same time retaining the readability of object manipulation code. The program `complex.cpp` illustrates the overloading of the + operator in the class template `complex`. In this case, the members of the complex number (real and imaginary) can be any of the standard data types.

```
// complex.cpp: template class for operator overloaded complex class
#include <iostream.h>
template <class T>
class complex
{
   private:
      T real;        // real part of complex number
      T imag;        // imaginary part of complex number
   public:
      complex()      // no argument constructor
      {
         real = imag = 0.0;
      }
      void getdata() // read complex number
      {
         cout << "Real Part ? ";
         cin >> real;
         cout << "Imag Part ? ";
         cin >> imag;
      }
      complex operator + ( complex c2 );  // complex addition
      void outdata( char *msg )           // display complex number
      {
         cout << msg << "(" << real;
         cout << ", " << imag << ")" << endl;
      }
};
template <class T>
complex <T> complex<T>::operator + ( complex <T>c2 )
{
   complex <T> temp;                  // object temp of complex class
   temp.real = real + c2.real;        // add real parts
   temp.imag = imag + c2.imag;        // add imaginary parts
   return( temp );                    // return complex object
}
void main()
{
   complex <int> c1, c2, c3;  // integer complex objects
   cout << "Addition of integer complex objects..." << endl;
   cout << "Enter complex number c1 .." << endl;
   c1.getdata();
```

```
   cout << "Enter complex number c2 .." << endl;
   c2.getdata();
   c3 = c1 + c2;       // integer addition
   c3.outdata("c3 = c1 + c2: " );          // display result
   complex <float> c4, c5, c6;   // integer complex objects
   cout << "Addition of float complex objects..." << endl;
   cout << "Enter complex number c4 .." << endl;
   c4.getdata();
   cout << "Enter complex number c5 .." << endl;
   c5.getdata();
   c6 = c4 + c5;  // floating addition
   c6.outdata("c6 = c4 + c5: " );          // display result
}
```

Run

```
Addition of integer complex objects...
Enter complex number c1 ..
Real Part ? 1
Imag Part ? 2
Enter complex number c2 ..
Real Part ? 3
Imag Part ? 4
c3 = c1 + c2: (4, 6)
Addition of float complex objects...
Enter complex number c4 ..
Real Part ? 1.5
Imag Part ? 2.5
Enter complex number c5 ..
Real Part ? 2.4
Imag Part ? 3.7
c6 = c4 + c5: (3.9, 6.2)
```

In `main()`, the statements

```
complex <int> c1, c2, c3;      // integer complex objects
complex <float> c4, c5, c6;    // integer complex objects
```

when encountered by the compiler, it creates two complex classes internally for handling numbers with integer and real data type members and instances of those classes. The statement

```
c3 = c1 + c2;       // integer addition
```

performs integer operation on complex objects, and the statement

```
c6 = c4 + c5;       // floating addition
```

performs floating-point operation on complex objects.

Review Questions

16.1 What is generic programming ? What are its advantages and state some of its applications ?

16.2 What is a function template ? Write a function template for finding the largest number in a given array. The array parameter must be of generic-data types.

16.3 Explain how the compiler processes calls to a function template.

16.4 State whether the following statements are TRUE or FALSE. Give reasons.

(a) generic-data type is known at runtime.
(b) function templates requires more memory space than normal function.
(c) templates are processed by the compiler.
(d) Special mechanism is required to execute function templates.
(e) The compiler reports an error if any one of the generic data-type indicated in template-type list is unused for defining formal parameters.
(f) A derived class of a template-based base class is not necessarily template derived class.
(g) Overloaded operator functions can be function templates.
(h) The syntax for defining objects of a class template is slightly different from the definition of the normal class's objects.
(i) Parameters to constructors can be of template type.

16.5 What is a class template ? Explain the syntax of a class template with suitable examples.

16.6 Explain how the compiler processes calls to a class template ?

16.7 Explain the syntax for inheriting template-based superclass. Note that the derived class can again be a template-based or non-template-based. Illustrate with suitable programming examples.

16.8 Write a template-based program for adding objects of the `Vector` class. Use dynamic data members instead of arrays for storing vector elements.

16.9 Write a program for manipulating linked list supporting node operations as follows:

```
node = node + 2;  node = node - 3;
Node <int> *n = node1 + node2;
```

The first statement creates a new node with node information 2 and the second statement deletes a node with node information 3. The node class must be of type template.

16.10 Write an interactive program for creating doubly linked-list. The program must support ordered insertion and deletion of a node. The doubly linked-list class must be of template type.

16.11 Design template classes such that they support the following statements:

```
Rupee <float> r1, r2;
Dollar <float> d1, d2;
d1 = r2;  // converts  rupee (Indian currency) to dollar (US currency)
r2 = d2;  // converts dollar (US currency) to rupee (Indian currency)
```

Write a complete program which does such conversions according to the world market value.

16.12 Consider an example of book shop which sells books and video tapes.It is modeled by `book` and `tape` classes. These two classes are inherited from the base class called `media`. The `media` class has common data members such as `title` and `publication`. The `book` class has data members for storing a number of pages in a book and the `tape` class has the playing time in a tape. Each class will have member functions such as `read()` and `show()`. In base class, these members have to be defined as virtual functions. Write a program which models this class hierarchy and processes their objects using pointers to base class only. (Use virtual functions and all classes must be template-based.)

17

Streams Computation with Console

In general, there are several kinds of streams to form physical entities such as streams of water (rivers), streams of electrons (electricity), streams of cars (traffic), and streams of characters (message packet). The notion of streams and streams computation can be visualized through the illustration of a river. It may be the Amazon river flowing into the Atlantic ocean as shown in Figure 17.1. Drops of water collectively form a continuous stream. Streams join to form a river. Looking over the upper river area to the lower river area, streams converge into one stream so that a tree of streams is formed, whose root stream goes into the ocean. One drop from one branch stream may reach the ocean a slightly earlier or later than another in a different branch stream.

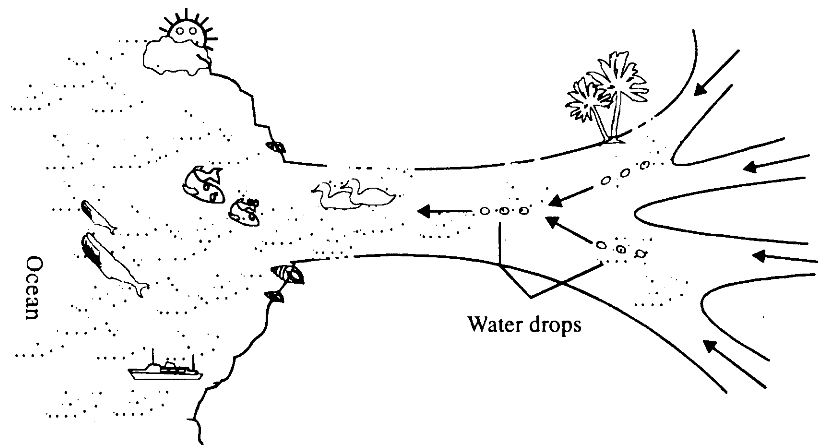


Figure 17.1: Streams of water drops flowing into ocean

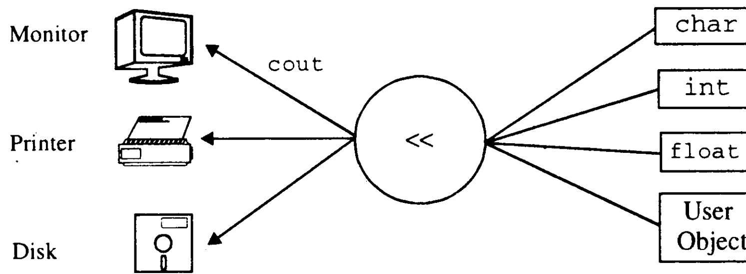
17.1 What are Streams ?

Every program must go through the process of *input-computation-output* flow so that it can take some data as input and generate the processed data as output. It necessitates the need for a mechanism, which supplies the input data to a program and presents the processed data in the desired form. In the earlier chapters, the input and output operations were performed using `cin` and `cout` with the stream operators >> and << respectively. Streams handling I/O operations are different from ANSI C functions. C++ supports a wide variety of features to control the way data is read and the output is presented.

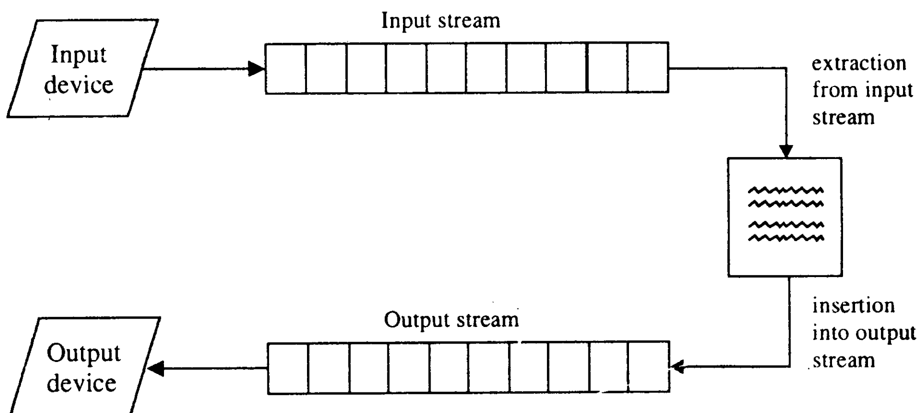
C++ uses the concept of streams and stream classes to perform I/O operations with console and disk files. The stream classes supporting console-based input and output operations are discussed in this chapter and those supporting file-based input and output operations are discussed in the next chapter, *Streams Computation with Files*.

C++ streams deal with a sequence of characters and hence, ocean in the above figure can be visualized as an object or a receiver and each drop of water as a character, flowing into the object.

Streams are classified into input streams and output streams. Streams resemble the *producer and consumer model*. The *producer* produces items to be consumed by the *consumer*. The producers and consumers are connected by the C++ operators >> or <<. In C++, the I/O system is designed to operate on a wide variety of devices including console, disks, printer etc. It is designed to provide a consistent and device independent interface. It allows uniform use of any I/O device—be it a disk, a terminal, or a printer as shown in Figure 17.2a. The computer resources involved in the stream computation include display, keyboard, files, printer, etc. The stream is an object flowing from one place to another. For instance, in nature, a stream normally refers to the flow of water from the hills to the oceans. Similarly, in C++, a stream is used to refer to the flow of data from a particular device to the program's variables. The device here refers to files, memory arrays, keyboard, console, and so on. In C++, these streams are treated as objects to support consistent access interface.



(a) Consistent stream interface with devices



(b) Data streams

Figure 17.2: C++ streams

Some of the above devices exhibit the characteristics of either a producer or a consumer and others exhibit the characteristics of both the producer and consumer depending on the operations performed on them. For instance, the keyboard exhibits the nature of only a producer; printer or monitor screen

exhibit the nature of only a consumer. Whereas, a file stored on the disk, can behave as a producer or consumer depending on the operation initiated on it. The stream model of C++ is shown in Figure 17.2b.

A stream is a series of bytes, which act either as a source from which input data can be extracted or as a destination to which the output can be sent. The source stream provides data to the program called the input stream and the destination stream that receives data from the program is called the output stream.

What are C++ Streams ?

The C language supports an extensive set of library functions for managing I/O operations. Every C programmer is familiar with `printf`, `scanf`, `puts`, `gets`, `fopen`, `fwrite`, `fread`, `fscanf`, `fclose`, and related I/O functions defined in the header file `stdio.h`. These functions have served programmers very well, but they are inadequate and clumsy when used with object-oriented programming. For instance, the user cannot add a new format either for `printf` or `scanf` function to handle the user-defined data type. Further, the `stdio.h` functions are inconsistent in parameter ordering and semantics.

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. Overloaded stream operators filter the user-defined data items and transfers only basic data items to the standard stream operators. Consider the following statements to illustrate the streams capability:

```
cout << complex1;

cin >> complex2;
```

The data-items `complex1` and `complex2` are the objects of the complex class. The operators >> or << do not know anything about the objects `complex1` and `complex2`. These are overloaded in the `complex` class as member functions, which process the attributes of complex objects as basic data-items. Collectively, it appears as if the stream operators operate even on objects of the complex class. This illusion is made possible because of the feature of overloading the stream operators.

The C++ language offers a mechanism which permits the creation of an extensible and consistent *input-output system* in the form of *streams library*. It is a collection of classes and objects which can be used to build a powerful system, or modified and extended to handle the user-defined data types. There are different classes for handling input and output streams, as also for streams connecting different devices to the program. C++ streams are also treated as filters, since they have capability to change the data representation from one number system to another when requested.

17.2 Predefined Console Streams

C++ contains several predefined streams that are opened automatically when the execution of a program starts. The most prominent predefined streams in C++ are related to the console device. The four standard streams `cin`, `cout`, `cerr`, and `clog` are automatically opened *before* the function `main()` is executed; they are closed *after* `main()` has completed. These predefined stream objects (are declared in `iostream.h`) have the following meaning:

`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 stream objects `cin` and `cout`, have been used extensively in the earlier chapters. It is known that `cin` (console input) represents the input stream connected to the standard input device and `cout` (console output) represents the output stream connected to the standard output device. The standard input and output devices normally refer to the keyboard and the monitor respectively. However, if required, these streams can be redirected to any other devices or files.

Comparison of I/O using C's stdio.h and C++'s iostream.h

The functions declared in the header file, `stdio.h` such as `printf`, `scanf`, etc., require the use of format strings. Consider an example of displaying the contents of the integer variable on the console to illustrate the flexibility offered by the C++ streams. If the variable `i` were to be defined by the statement

```
int i;
```

then the `printf` statement to display the value of the variable `i` would be,

```
printf("%d", i);
```

and the statement to read data would be,

```
scanf("%d", &i);
```

Consider a situation in which the `printf` or `scanf` statement occurs at several places in a program. Suppose the program specifications are changed, and it is decided that the variable `i` must hold larger values, the definition of `i` would be changed to,

```
long i;
```

The user is now left with the thankless job of searching for all the statements that read or display the variable `i` and replacing `%d` by `%ld` in the format strings. On the other hand, in C++, the `iostream.h` functions are overloaded to take care of all the basic types. For instance, the statements

```
cout << i;
cin >> i;
```

will work correctly without the need for any modification irrespective of the data type of the `i` variable. The stream based I/O operations can be performed with variables of all the basic data types such as `char`, `signed char`, `short int`, `long`, etc. In addition to these, the << and >> operators are overloaded to operate on pointers to characters also (for performing input or output with the NULL terminated strings). The traditional beginner's C program is usually called "Hello World" and is listed in the program `hello.c`.

```
/* hello.c: printing Hello World message */
#include <stdio.h>
void main()
{
  printf( "Hello World" );
}
```

Run

```
Hello World
```

The standard function `printf()` is in the C library that sends characters to the standard output device. The *Hello World* program will also work in C++, because C++ supports ANSI-C function library. A new C++ program that does the same operation as C's *Hello World* is listed in `hello.cpp`.

```
// hello.cpp: printing Hello World message
#include <iostream.h>
void main()
{
   cout << "Hello World";
}
```

Run

```
Hello World
```

The header file, `iostream.h` supports streams programming features by supporting predefined stream objects. The C++'s stream insertion operator, << sends the message `Hello World` to the predefined console object, `cout` which, in turn, prints on the console.

Output Redirection

The output generated by `cout` can be redirected to files whereas, that generated by `cerr` and `clog` cannot be redirected. That is, the following on the command line,

```
shell: hello > outfile
```

redirects console output to the file named `outfile`. The output file contains only those messages generated by `cout` but not by `cerr` and `clog`. They always redirect to console as illustrated in the program `redirect.cpp`.

```
// redirect.cpp: printing Hello World message
#include <iostream.h>
void main()
{
   cout << "Hello World with cout\n";
   cerr << "Hello World with cerr\n";
   clog << "Hello World with clog\n";
}
```

Run

```
Hello World with cerr
Hello World with clog
```

Note: The program is executed by issuing the following command at the shell prompt:

```
redirect > outfile
```

On execution, the messages shown at RUN appear on the console whereas the first message `Hello World with cout` is stored in the file `outfile`.

The main advantage of using `iostream.h` functions over the `stdio.h` functions is data-independence; the freedom to write code without worrying too much about the variable types. Mixed usage of `stdio` and the `stream` class functions to perform output is not advisable. This is because they use different buffers and the order in which the output appears may not conform to the order in which the output statements appear in the program.

Features of cin and cout

Before examining the facilities available with `cout` and `cin`, it is useful to know that the objects `cin` and `cout` are instances of certain classes defined in `iostream.h`. The object `cout` is an instance of

class ostream_withassign, which is derived from the superclass ostream. Hence, effectively cout has the functionality of the class ostream. Similarly, cin an instance of the class istream_with_assign has the functionality of the class istream.

17.3 Hierarchy of Console Stream Classes

The C++ input-output system supports a hierarchy of classes that are used to manipulate both the console and disk files, called stream classes. The stream classes are implemented in a rather elaborate hierarchy. The knowledge of C++'s input and output stream class hierarchy will result in the potential utilization of stream classes. Figure 17.3, depicts hierarchy of classes, which are used with the console device.

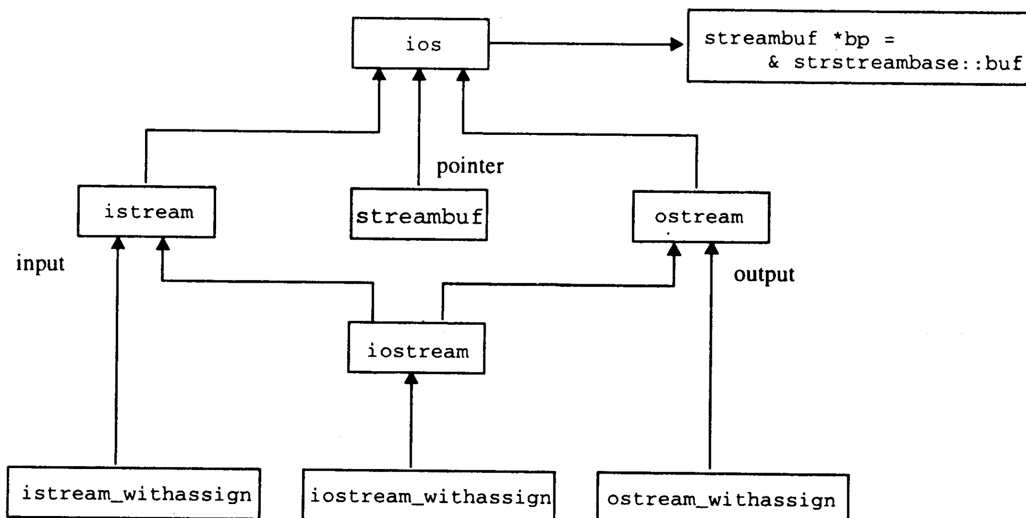


Figure 17.3: Hierarchy of console stream classes

The iostream facility of C++ provides an easy means to perform I/O. The class istream uses the predefined stream cin that can be used to read data from the standard input device. The extraction operator >>, is used to get data from a stream. The insertion operator <<, is used to output data into a stream. A stream object must appear on the left side of the << or >> operator; however, multiple stream operators can be concatenated on a single line, even when they refer to objects of different types. For instance, consider the following statements:

```
cout << item1 << "***" << c1 << my_object << 22;
cin >> int_var >> float_var >> my_object;
```

The first statement outputs objects of different types (both the standard and user defined) and the second statement reads data of different types.

The classes istream, ostream, and iostream, which are designed exclusively to manage the console device, are declared in the header file iostream.h. The actions performed by these classes related to console device management are described below:

ios class: It provides operations common to both input and output. It contains a pointer to a buffer object (streambuf). It has constants and member functions that are essential for handling formatted input and output operations.

The classes derived from the ios class (istream, ostream, iostream) perform specialized input-output operations with high-level formatting:

- istream (input stream) does formatted input.
- ostream (output stream) does formatted output.
- iostream (input/output stream) does formatted input and output.

The pointer streambuf in the ios class provides an abstraction for communicating to a physical device and classes derived from it deal with files, memory, etc. The class, ios communicates to a streambuf, which maintains information on the state of the streambuf (good, bad, eof, etc.), and maintains flags used by the istream and ostream.

istream class: It is a derived class of ios and hence inherits the properties of ios. It defines input functions such as get(), getline(), and read(). In addition, it has an overloaded member function, stream extraction operator >>, to read data from a standard input device to the memory items.

ostream class: It is a derived class of ios, and hence, inherits the properties of ios. It defines output functions such as put() and write(). In addition, it has an overloaded member function, stream insertion operator <<, to write data from memory items to a standard output device.

iostream class: It is derived from multiple base classes, istream and ostream, which are in turn inherited from the class ios. It provides facility for handling both input and output streams, and supports all the operations provided by istream and ostream classes.

The classes istream_withassign, ostream_withassign, and iostream_withassign add the assignment operators to their parent classes.

17.4 Unformatted I/O Operations

The most commonly used objects throughout all C++ programs are cin and cout. They are predefined in the header file, iostream.h, which supports the input and output of data of various types. This is achieved by overloading the operators << and >> to recognize all the basic data types. The input or extraction operator is overloaded in the istream class and output or insertion operator is overloaded in the ostream class.

put() and get() Functions

The stream classes of C++ support two member functions, get() and put(). The function get() is a member function of the input stream class istream and is used to read a single character from the input device. The function put() is a member function of the output stream class ostream and is used to write a single character to the output device. The function get() has two versions with the following prototypes:

```
void get( char & );
int get(void);
```

Both the functions can fetch a white-space character including the blank space, tab, and newline character. It is well known that, the member functions are invoked by their objects using dot operators. Hence, these two functions can be used to perform input operation either by using the predefined

object, `cin` or an user defined object of the `istream` class. The program `get.cpp` illustrates the use of `get()` function to read a line (until carriage return key is pressed).

```
// get.cpp: Read characters using get() of istream
#include <iostream.h>
void main()
{
   char c;
   cin.get( c );
   while( c != '\n' )
   {
      cout << c;
      cin.get( c ); // reads a character
      // replace the above statement by cin >> c; and see the output
   }
}
```

Run

```
Hello World
Hello World
```

In `main()`, the statement

```
cin.get( c );
```

invokes the member function `get()` of the object `cin` of the `istream` class. It reads a character into the variable c from the standard input device. If this statement is replaced by the statement,

```
cin >> c;
```

it will not work as desired, since the operator >> will skip blanks and newline characters. Another version of `get()` can also be used in the above program as follows:

```
c = cin.get();
```

It reads a single character and returns the same.

The function `put()`, which is a member function of the output stream class `ostream` prints a character representation of the input parameter. For instance, the statement,

```
cout.put( 'R' );
```

prints the character `R`,and the statement

```
cout.put( c );
```

prints the contents of the character variable `c`. The input parameter can also be a numeric constant and hence, the statement

```
cout.put( 65 );
```

prints the character `A` (65 is a ASCII code of character A). The program `put.cpp` prints the ASCII table (since `put()` considers input parameter as a ASCII code of a character to be printed.)

```
// put.cpp: prints ASCII table using put() function
#include <iostream.h>
void main()
{
   char c;
   for( int i = 0; i < 255; i++ )
   {
```

```
        if( i == 26 )
           continue;
        cout << i << " ";
        cout.put( i );    // change to cout << i; and see the output difference
        cout << endl;
    }
}
```

Run

[prints ASCII code and its character representation]

In `main()`, the statement

```
    cout.put( i );
```

prints a character represented by the ASCII code whose value is passed as an input argument through the variable `i`.

`getline()` and `write()` Functions

The C++ stream classes support line-oriented functions, `getline()` and `write()` to perform input and output operations. The `getline()` function reads a whole line of text that ends with new line or until the maximum limit is reached. Consider the program `space1.cpp` for reading an input string having a blank space in between.

```
// space1.cpp: the effect of white-space characters on the >> operator
#include <iostream.h>
#include <iomanip.h>
void main()
{
   char test[40];
   cout << "Enter string: ";
   cin >> test;
   cout << "Output string: ";
   cout << test;
}
```

Run

```
Enter string: Hello World
Output string: Hello
```

In `main()`, the statement

```
    cin >> test;
```

reads a string until it encounters a white space. If the input to the above program is `"Hello World"`, the output is going to be just `"Hello"`. The reason being the operator >> considers all white-space characters in the input stream as delimiters. To remedy this, use the member function `getline()` of the `cin` object's class as shown in the program `space2.cpp`.

```
// space2.cpp: the effect of white-space characters on the >> operator
#include <iostream.h>
#include <iomanip.h>
void main()
{
```

```
   char test[40];
   cout << "Enter string: ";
   cin.getline( test, 40 );
   cout << "Output string: ";
   cout << test;
}
```

Run

```
Enter string: Hello World
Output string: Hello World
```

In `main()`, the statement

```
cin.getline( test, 40 );
```

reads a string until it encounters the new line character or maximum number of characters (40). Now, an input of "Hello World" will produce the output as desired. The `istream::getline` member function has the following versions:

```
istream& getline(signed char*, int len, char = '\n');
istream& getline(unsigned char*, int len, char = '\n');
```

They operate in the following ways:

- extracts character up to the delimiter
- stores the characters in the buffer
- removes the delimiter from the input stream
- does not place the delimiter into the buffer
- maximum number of characters extracted is `len-1`

The terminator character can be any character. The terminator character is *read but not saved* into a buffer; instead, it is replaced by the null character.

The prototype of `write()` functions is:

```
ostream::write( char * buffer, int size );
```

It displays `size` (second parameter) number of characters from the input buffer. The display does not stop even when the NULL character is encountered; If the length of the buffer is less than the indicated size, it displays beyond the bounds of buffer. Therefore, it is the responsibility of the user to make sure that the size does not exceed the length of the string. The program `stand.cpp` illustrates the use of write in string processing.

```
// stand.cpp: display stand of "Object Computing with C++";
#include <iostream.h>
#include <string.h>
void main()
{
   char *string1 = "Object-Computing";
   char *string2 = " with C++";
   int i;
   int len1 = strlen( string1 );
   int len2= strlen( string2 );
   for( i = 1; i < len1; i++ )
   {
      cout.write( string1, i );
      cout << endl;
```

```
    }
    for( i = len1; i > 0; i-- )
    {
        cout.write( string1, i );
        cout << endl;
    }
    // print both the string
    cout.write( string1, len1 );
    cout.write( string2, len2 );
    cout << endl;
    // above two write() can be replaced below single statement
    cout.write( string1, len1 ).write( string2, len2 );
    cout << endl;
    cout.write( string1, 6 );
}
```

Run

```
O
Ob
Obj
Obje
Objec
Object
Object-
Object-C
Object-Co
Object-Com
Object-Comp
Object-Compu
Object-Comput
Object-Computi
Object-Computin
Object-Computing
Object-Computin
Object-Computi
Object-Comput
Object-Compu
Object-Comp
Object-Com
Object-Co
Object-C
Object-
Object
Objec
Obje
Obj
Ob
O
Object-Computing with C++
Object-Computing with C++
Object
```

In main(), the last statement

```
cout.write( string1, 6 );
```

indicates to display six characters from the string, string1 even though the input string has more characters than the number of characters requested to be displayed. The two statements,

```
cout.write( string1, len1 );
cout.write( string2, len2 );
```

can be replaced by the single statement,

```
cout.write( string1, len1 ).write( string2, len2 );
```

The *dot* operator with the predefined object cout indicates that the function write is a member of the class ostream. The invocation of write() function returns the object of type ostream which again invokes the write() function.

17.5 Formatted Console I/O Operations

Most programs need to output data in various styles. A common requirement is to reserve an area of the screen for a field, without knowing the number of characters the data of that field will occupy. To do this, there must be a provision for alignment of fields to left or right, or padded with some characters. C++ supports a wide variety of features to perform input or output in different formats. They include the following:

- ios stream class member functions and flags
- Standard manipulators
- User-defined manipulators

ios Class Functions and Flags

The stream class, ios contains a large number of member functions to assist in formatting the output in a number of ways. The most important among these functions are shown in Table 17.1.

Function	Task Performed
width()	Specifies the required number of fields to be used while displaying the output value.
precision()	Specifies the number of digits to be displayed after the decimal point.
fill()	Specifies a character to be used to fill the unused area of a field. By default, fills blank space character.
setf()	Sets format flag that control the form of output display
unsetf()	Clears the specified flag

Table 17.1: ios class member functions

Defining Display Field Width

The function width() is a member function of the ios class and is used to define the width of the field to be used while displaying the output value. It must be accessed using objects of the ios class

(commonly accessed using `cout` object). It has the following two forms:

```
int width();
int width(int w);
```

where w is the field width i.e., number of columns to be used for displaying output. The first form of `width()` returns the current width setting whereas, the second form `width(int)` sets the width to the specified integer value and returns the previous width. It specifies field width for the item, which is displayed first immediately after the setting. After displaying an item, it will revert to the default width. For instance, the statements

```
cout.width( 4 );
cout << 20 << 123;
```

produce the following output:

		2	0	1	2	3

The first value is printed in right-justified form in four columns. The next item is printed immediately after first item without any separation; `width(4)` is then reverted to the default value, which prints in left-justified form with default size. It can be overcome by explicitly setting width of every item with each `cout` statement as follows:

```
cout.width( 4 );
cout << 20;
cout.width( 4 );
cout << 123;
```

These statements produce the following output.

		2	0		1	2	3

It should be noted that field width should be specified for each item independently if a width other than the default is desired for output. If the field width specified is smaller than the required width to display items, the field is expanded to the required space without truncation. For instance,

```
cout.width( 2 );
cout << 2000;
```

These statements produce the following output:

2	0	0	0

without truncating eventhough width is specified as two. The program `student.cpp` illustrates the use of width function in formatting the displayed output.

```
// student.cpp: printing student details in the form of table
#include <iostream.h>
const int MAX_MARKS = 600;     // maximum marks
class student
{
   private:
      char name[11];        // name of a student
```

```
      int marks;            // marks scored by a student
   public:
      void read();
      void show();
};
void student::read()
{
   cout << "Enter Name: ";
   cin >> name;
   cout << "Enter Marks Secured: ";
   cin >> marks;
}
void student::show()
{
   cout.width( 10 );
   cout << name;
   cout.width( 6 );
   cout << marks;
   cout.width( 10 );
   cout << int(float(marks)/MAX_MARKS * 100);   // percentage
}
void main()
{
   int i, count;
   student *s;    // pointers to objects
   cout << "How many students ? ";
   cin >> count;
   s = new student[count];    // array of objects, student s[count]
   for( i = 0; i < count; i++ )
   {
      cout << "Enter Student " << i+1 << " details.." << endl;
      s[i].read();
   }
   cout << "Student Report..." << endl;
   cout.width( 3 );
   cout << "R#";
   cout.width( 10 );
   cout << "Student";
   cout.width( 6 );
   cout << "Marks";
   cout.width( 15 );
   cout << "Percentage" << endl;
   for( i = 0; i < count; i++ )
   {
      cout.width( 3 );
      cout << i+1;   // roll_no
      s[i].show();
      cout << endl;
   }
}
```

Run

```
How many students ? 3
Enter Student 1 details..
Enter Name: Tejaswi
Enter Marks Secured: 450
Enter Student 2 details..
Enter Name: Rajkumar
Enter Marks Secured: 535
Enter Student 3 details..
Enter Name: Bindu
Enter Marks Secured: 429
Student Report...
 R#    Student    Marks    Percentage
  1    Tejaswi    450      75
  2    Rajkumar   525      87
  3    Bindu      429      71
```

Setting Precision

The function `precision()` is a member of the `ios` class and is used to specify the number of digits to be displayed after the decimal point while printing a floating-point number. By default, the precision size is six. This function must be accessed using objects of the ios class (commonly accessed using `cout` object). It has the following two forms:

```
int precision();            // returns current precision
int precision(int d);
```

where `d` is the number of digits to the right of the decimal point. It sets the floating-point precision and returns the previous setting. For example, the statements

```
cout.precision( 2 );
cout << 2.23 << endl;
cout << 5.169 << endl;
cout << 3.5055 << endl;
cout << 4.003 << endl;
```

will produce the following output:

```
2.23          (perfect fit)
5.17          (rounded)
3.51          (rounded)
4             (no trailing zeros, truncated)
```

After displaying an item, the user defined precision will not revert to the default value. Different values can be processed with different precision by having multiple precision statements. For instance,

```
cout.precision( 1 );
cout << 2.23 << endl;
cout.precision( 3 );
cout << 5.1691 << endl;
```

will produce the following output:

```
2.2       (truncated)
5.169     (truncated)
```

Consider the statements:

```
cout.precision(3);
```

```
cout << 12.53 << 20.5 << 2;
```

which produce the following output all packed together:

1	2	.	5	3	2	0	.	5	2

It can be overcome by the combined use of `width()` and precision to control the output format. The statements

```
cout.precision( 2 );
cout.width( 6 );
cout << 12.53;
cout.width( 6 );
cout << 20.5;
cout.width( 6 );
cout << 2;
```

will produce the following output:

	1	2	.	5	3			2	0	.	5						2

It must be noted from the above output that the unused width is filled with blank characters. **Unlike** `width()`, the `precision()` must be reset for each data item being output if new **precision is** desired.

Filling and Padding

The function `fill()` is a member of the `ios` class and is used to specify the character to be **displayed** in the unused portion of the display width. By default, blank character is displayed in the unused portion if the display width is larger than that required by the value. It has the following two forms:

```
int fill();          // returns current fill character
int fill( ch );
```

where `ch` is the character to be filled in the unused portion. For example, the statements

```
cout.fill( '*' );
cout.precision( 2 );
cout.width( 6 );
cout << 12.53;
cout.width( 6 );
cout << 20.5;
cout.width( 6 );
cout << 2;
```

will produce the following output:

*	1	2	.	5	3	*	*	2	0	.	5	*	*	*	*	*	2

It is seen from the above output that the unused width is filled with asterisk character as set by the statement `cout.fill('*');`. Similar to `precision()`, the effect of `fill()` continues unless explicitly modified by the other `fill()` statement. It is illustrated by the program `salary.cpp`.

```
// salary.cpp: filling and padding
#include <iostream.h>
void main()
{
   char *desig[5] = { "CEO", "Manager", "Receptionist", "Clerk", "Peon" };
   int salary[5] = { 10200, 5200, 2950, 950, 750 };
   cout << "Salary Structure Based on Designation" << endl;
   cout << "-------------------------------------" << endl;
   cout.width( 15 );
   cout << "Designation";
   cout << "     ";
   cout.width( 15 );
   cout << "Salary (in Rs.)" << endl;
   cout << "-------------------------------------" << endl;
   for( int i = 0; i < 5; i++ )
   {
      cout.fill('.');
      cout.width(15);
      cout << desig[i];
      cout << "     ";
      cout.fill('*');
      cout.width(15);
      cout << salary[i] << endl;
   }
   cout << "-------------------------------------" << endl;
}
```

Run

```
Salary Structure Based on Designation
-------------------------------------
    Designation    Salary (in Rs.)
-------------------------------------
............CEO    **********10200
........Manager    ***********5200
...Receptionist    ***********2950
..........Clerk    ************950
...........Peon    ************750
-------------------------------------
```

Note that such a form of output representation is extensively used by financial institutions to represent money transactions so that no one can modify the amount (money representation) easily.

Formatting with Flags and Bit-fields

From the earlier examples, it can be noted that, when the function `width()` is used, results are printed in the right-justified form (which is not a usual practice). C++ provides a mechanism to set the printing of results in the left-justified form, scientific notation etc. The member function of the `ios` class, `setf()` (setf stands for set flags) is used to set flags and bit-fields that control the output. It has the following two forms:

```
long setf(long _setbits, long _field);
long setf(long _setbits );
```

where `_setbits` is one of the flags defined in the class `ios`. It specifies the format action required for

the output, and `_field` specifies the group to which the formatting flag belongs. Both the forms return the previous settings. The flags, bit-fields when set with `setf()` and their actions is shown in Table 17.2. There are three bit-fields and each group has format flags that are mutually exclusive. For instance,

```
cout.setf( ios::right, ios::adjustfield );
cout.setf( ios::oct, ios::basefield );
cout.setf( ios::scientific, ios::floatfield );
```

Note that the flag argument (first) should be one of the group (bit-field) of the second argument.

Flags value	Bit field	Effect produced
ios::left	ios::adjustfield	Left-justified output
ios::right	ios::adjustfield	Right-adjust output
ios::internal	ios::adjustfield	Padding occurs between the sign or base indicator and the number, when the number output fails to fill the full width of the field.
ios::dec	ios::basefield	Decimal conversion
ios::oct	ios::basefield	Octal conversion
ios::hex	ios::basefield	Hexadecimal conversion
ios::scientific	ios::floatfield	Use exponential floating notation
ios::fixed	ios::floatfield	Use ordinary floating notation

Table 17.2: Flags and bit fields for setf function

Consider the following statements:

```
cout.setf( ios::left, ios::adjustfield );
cout.fill( '*' );
cout.precision( 2 );
cout.width( 6 );
cout << 12.53;
cout.width( 6 );
cout << 20.5;
cout.width( 6 );
cout << 2;
```

The output produced by the above statements is:

1	2	.	5	3	*	2	0	.	5	*	*	2	*	*	*	*	*

The statements

```
cout.setf( ios::internal, ios::adjustfield );
cout.fill( '*' );
cout.precision( 3 )·
```

```
cout.width( 10 );
cout << -420.53;
```

will produce the following output:

-	*	*	*	4	2	0	.	5	3

If the last statement is replaced by,

```
cout << -420.534;
```

the following output will be generated:

-	*	*	4	2	0	.	5	3	4

Note that the sign is left justified and the value is right justified. The space between them is filled with stars.

Displaying Trailing Zeros and Plus Sign

Streams support the feature of avoiding truncation of the trailing zeros in the output. For instance, the following statements:

```
cout << 20.55 << endl;
cout << 55.40 << endl;
cout << 10.00 << endl;
```

produce the output as shown below:

2	0	.	5	5
5	5	.	4	
1	0			

It can be observed that the trailing zeros in second and third output have been truncated. The `ios` class has the flag, showpoint which when set, prints the trailing zeros also. It is set by the following statement

```
cout.setf( ios::showpoint );
```

which causes the `cout` to display the trailing decimal point and zero. The following statements

```
cout.setf( ios::showpoint );
cout.precision( 2 );
cout << 20.55 << endl;
cout << 55.40 << endl;
cout << 10.00 << endl;
```

would produce the output as shown below:

2	0	.	5	5
5	5	.	4	0
1	0	.	0	0

Similarly, the plus symbol can be printed using the following statement:

```
cout.setf( ios::showpos );
```

For example, the statements

```
cout.setf( ios::showpos );        // positive sign
cout.setf( ios::showpoint );      // trailing zero and point
cout.setf( ios::internal, ios::adjustfield );
cout.precision( 3 );
cout.width( 10 );
cout << 420.53;
```

will produce the following output:

+			4	2	0	.	5	3	0

Table 17.3 presents summary of flags that do not have bit fields for the `setf` function.

Flag's value	Effect produced
`ios::showbase`	Use base indicator on output
`ios::showpos`	Add '+' to positive integers
`ios::showpoint`	Include decimal point and trailing zeros in output
`ios::uppercase`	Upper-case hex output
`ios::skipws`	Skips white-space characters on input.
`ios::unitbuf`	Flush after insertion. (i.e., use a buffer of size 1)
`ios::stdio`	Flush stdout and stderr after insertion

Table 17.3: Flags that do not have bit fields for setf function

The flag setting `ios::skipws` is set by default. The white-space characters are space, tab, newline, carriage return, form feed and vertical tab. While performing formatted input (with the >> operator), an input stream (such as `cin`) behaves as if these characters are not present in the input. Use this flag with the `resetiosflags` manipulator, to prevent skipping white-space characters.

The flags can be reset by using the `ios::unsetf` member function. It has the following syntax:

```
long unsetf(long);
```

and is invoked as follows:

```
cout.unsetf( ios::showpos );
```

It clears the bits corresponding to show positive-sign symbol (when number displayed is positive) and returns the previous settings.

17.6 Manipulators

The C++ streams package makes use of the notion of stream manipulators, principally as a means of manipulating the formatting state associated with a stream. These manipulators are functions that can be used with the << or the >> operator to alter the behavior of any stream class instances including the

cin and cout. C++ has manipulators which produce output and consume input to extend stream I/O formatting. Such manipulators can be especially useful for simple parsing of stream inputs. Manipulators are broadly categorized as producers and consumers. A producer manipulator is one which generates output on an output stream, for example, endl. Similarly, a consumer manipulator consumes input from an input stream, for example, ws.

Manipulators are special functions that are specifically designed to modify the working of a stream. They can be embedded in the I/O statements to modify the form parameters of a stream. All the predefined manipulators are defined in the header file iomanip.h. Manipulators are more convenient to use than their counterparts, defined by the ios class. There can be more than one manipulator in a statement and they can be chained as shown in the following statements:

```
cout << manip1 << manip2 << manip2 << item;
cout << manip1 << item1 << item2 << manip2 << item3;
```

This kind of chaining of manipulators is useful in displaying several columns of output. Manipulators are categorized into the following two types:

- Non-Parameterized Manipulators
- Parameterized Manipulators

As mentioned before, cout and cin work elegantly with any basic type. They do not require specification of type of variables while performing I/O. The format string of C's I/O function requires display control information such as width, number system, etc., apart from the variable types in the format string. The program hex.c clarifies these concepts.

```
/* hex.c: read hexadecimal number and display the same in decimal */
#include <stdio.h>
void main()
{
  int num;
  printf( "Enter any hexadecimal number: " );
  scanf( "%x", &num );      /*Input in hexadecimal*/
  /*output i in decimal,in a field of width 6*/
  printf( "The input number in decimal = " );
  printf( "%6d", num);
}
```

Run

```
Enter any hexadecimal number: ab
The input number in decimal =    171
```

This kind of code is often useful. The question arises—*How can this be done with cin and cout ?* The answer lies in the manipulators. For example, the above lines of code that used scanf and printf can be rewritten as listed in the program hex.cpp.

```
// hex.cpp: read hexadecimal number and display the same in decimal
#include <iostream.h>
#include <iomanip.h>    // for manipulators
void main()
{
  int num;
  cout << "Enter any hexadecimal number: ";
```

```
  cin >> hex >> num;     // Input in hexadecimal
  // output i in decimal,in a field of width 6
  cout << "The input number in decimal = ";
  cout << setw( 6 ) << num;
}
```

Run

```
Enter any hexadecimal number: ab
The input number in decimal =    171
```

The manipulator `hex` sets the conversion base for `cin` to 16. So `cin` interprets the input characters as digits of a hexadecimal number. The manipulator `setw` sets the field width as 6 for `cout`. Thus, the input to the above program (`ab`) is converted into decimal and displayed (16*a+b = 16*10+11 = 171).

The C++ `iostream` package contains a rather small handful of predefined producer consumer manipulators, the only instance of consumer being the white-space eater, for example, `ws`. Other predefined manipulators set stream state variables which influence processing of input and output, for example, hex. The implementation of the `ios` class as well as the implementation of the insertion and extraction operators correspond to the data type of an item they process. The list of non-parameterized manipulators and parameterized manipulator functions are shown in Table 17.4 and 17.5 respectively. Each one of these can be used with either the << or the >> operator without incurring any compile-time errors. But some of them affect only output streams such as `cout`, and some others, only input streams such as `cin`. Unless otherwise mentioned, these manipulators affect both types of streams. The first six manipulators - `dec`, `hex`, `oct`, `ws`, `endl`, and `ends` are defined in `iostream.h` itself and the rest are in the header file `iomanip.h`.

Manipulator	Action Performed
`dec`	Sets the conversion base to 10
`hex`	Sets the conversion base to 16
`oct`	Sets the conversion base to 8
`ws`	Extracts white-space characters from an input stream. Characters in the stream will be extracted until a non-white-space character is found, or an error (such as EOF) occurs. As expected, it affects only input streams.
`endl`	Outputs a newline and flushes stream Affects only output streams "\n"
`ends`	Outputs a NULL character ('\0') Affects only output streams
`flush`	Flushes the stream. Affects only output streams

Table 17.4: C++'s predefined non-parameterized manipulators