Chapter: 14

Miscellaneous topics & STL

Namespaces (advanced), Conversion functions , Static class members, const, mutable, asm , Linkage specifiers, Array based I/O & STL (Standard Template Library)

14.1 *namespace* Details

Namespaces purpose is to ***localize the names of identifiers*** to avoid **name collisions**. Prior to the invention of namespaces, all of variable, function, and class names competed for slots in the ***global namespace***, and many conflicts arose. For example,

- ⇨ If your program defined a function called ***toupper()***, it could (depending upon its parameter list) override the standard library function ***toupper()*** because both names would be stored in the ***global namespace***.
- ➲ **Name collisions** were compounded when ***two or more third-party libraries*** were used by the ***same program***. In this case, it was possible-even likely-that a name defined by one library would conflict with the same name defined by another library.
- ➲ In early the entire C++ library was defined within the global namespace (which was, of course, the only namespace). Now, however, the C++ library is defined within its own namespace std, which reduces the chance of name collisions.

❒ The ***namespace*** localizes the visibility of names declared within it, a ***namespace*** allows the same name to be used in different contexts without giving rise to conflicts. You can also create your own ***namespaces*** within your program to localize the visibility of any names that you think might cause conflicts. This is especially important if you are creating **class or function libraries.**

☛ The ***namespace*** keyword allows you to partition the ***global namespace*** by creating a ***declarative region***. In essence, a ***namespace*** defines a ***scope***. The general form of ***namespace*** :

namespace name { */* declarations */* ***}***

➢ Anything defined within a ***namespace*** statement is **within the scope of that namespace**.

➢ Here is an example of a namespace:

```
namespace MyNameSpace {
                        int i, k;
                        void myfunc(int j){ cout << j; }
                        class myclass{ public :
                                              void seti(int x){ i = x; }
                                              int geti(){ return i; }
                                       };
                      }
```

▶ ***i, k, mufunc(),*** and the class ***myclass*** are part of the scope defined by the ***MyNameSpace namespace***.

❒ ***Scope resolution operator to access class/members inside namespace:*** Since ***namespace*** defines a ***scope***, you need to use the ***scope resolution operator*** to refer to **objects** declared **within** a ***namespace*** from **outside** that ***namespace***.

❖ ***Declaring object:*** To declare an **object** of type ***myclass*** from outside ***MyNameSpace***,

MyNameSpace :: myclass ob;

☛ To access a **member** of a namespace from **outside** its ***namespace***, precede the member's name with the name of the namespace followed by the ***scope resolution operator***.

❖ ***Accesing members:*** To assign the value ***10*** to ***i*** from code outside ***MyNameSpace***,

MyNameSpace :: i = 10;

☛ Identifiers ***declared within*** a ***namespace*** can be referred to directly within that ***namespace***. For example, in ***MyNameSpace*** the ***"return I"*** statement uses ***i*** directly.

❒ ***The keyword "using":*** Sometime it's a pain to use "**::**" for too many references. In that case we use the "***using***" keyword. The ***using*** statement has these two general forms:

using namespace name ;	**using name :: member ;**
In this form, ***name*** specifies the name of the ***namespace*** you want to access. When you use this form, **all of the members** defined within the specified namespace are brought into the current namespace and can be used **without qualification**.	In the second form, only a **specific member** of the ***namespace*** is made visible.
For example, assuming ***MyNameSpace*** as shown above, the following using statements and assignments are valid:	
using namespace ***MyNameSpace*** ; */* all members are visible */* i = 10; */* all members of MyNameSpace are visible */*	**Using namespace** ***MyNameSpace*** :: k; */*only k is made visible */* k = 10; */* OK because k is visible */*

❒ There can be ***more than one namespace*** declaration of the ***same name***. This allows a namespace to be split over **several *.cpp*** files or even separated within the **same *.cpp*** file. Consider the following example:

```
namespace NS { int i; }
// ... . . . .. .. .
namespace NS { int j; }
```

⇨ Here ***NS*** is split into two pieces. However, the contents of each piece are still within the **same namespace**, ***NS***.

❒ A ***namespace*** must be declared **outside of all other scopes**, with one exception: ***a namespace can be nested within another***. For example, you cannot declare namespaces that are localized to a function.

❒ There is a special type of ***namespace***, called an ***unnamed namespace***, that allows you to create ***identifiers that are unique within a file***. It has this general form:

namespace { */* declarations */* }

☛ i.e, within the ***.cpp*** file that contains the ***unnamed namespace***, the members of that ***namespace*** can be used ***directly***, without qualification. But outside the file, the identifiers are ***unknown***.

NOTE

You will not usually need to create namespaces for most small-to medium-sized programs. namespaces are useful for creating ***libraries of reusable code*** or if you want to ensure the widest Portability.

❒ ***Example 1:*** Here is an example that illustrates the attributes of a namespace.

```
using namespace std;

/* define a namespace */
namespace firstNS {
        class demo { int i;
                public :
                        demo(int x){i = x;}
                        void seti(int x){i = x;}
                        int geti(){return i;}          };
        char str[] = " Illustrating namespaces \n";
        int counter ;
        }
/* define another namespace */
namespace secondNS { int x, y; }

int main(){
firstNS :: demo ob(10) ;        /* Creating object, use scope resolution */
cout << " Value of ob is: " << ob.geti();     /* direct uses of object 'ob' */
cout << endl ;
ob.seti(99) ;
cout << " Value of ob is now : " << ob.geti();
cout << endl ;

using firstNS :: str;           /* bring str into current scope */
cout << str ;
                        /* work with both namespaes */

                /* bring all of firstNS into current scope */
using namespace firstNS ;
for ( counter = 10; counter ; counter --) cout << counter << " ";
cout << endl ;

                /* use secondNS namespace */
secondNS ::x = 10;              /* accessing and setting values */
secondNS ::y = 20;              /* accessing and setting values */
cout << "x y: " << secondNS ::x;
cout << ", " << secondNS ::y << endl ;

                /* bring another namespace into view */
using namespace secondNS ;
demo xob(x), yob(y);            /* using secondNS with firstNS */
cout << "xob , yob : " << xob.geti() << ", ";
cout << yob.geti() << endl ;
return 0; }
```

OUTPUT :

```
Value of ob is : 10
Value of ob is now : 99
Illustrating namespaces
10 9 8 7 6 5 4 3 2 1
x, y: 10, 20
xob, yob: 10, 20
```

☛ Notice, once object "ob" has been declared, its member functions can be used without namespace qualification.

☛ The program illustrates one important point: **using one namespace does not override another**. When you ***bring a namespace into view***, it simply adds its names to whatever other namespaces are currently in effect. Thus, by the end of this program the ***std*** [C++'s standard library, 1st line], ***firstNS***, and ***secondNS*** namespaces have been added to the ***global namespace***.

❒ ***Example 2:*** Namespace can be split between ***.cpp*** files or within a single ***.cpp*** file; its contents are additive.

```
namespace Demo{ int a; }          /*  a is In Demo namespace */
int x;                            /* x is in global namespace */
namespace Demo{ int b; }          /* b is in Demo namespace , too */
int main(){   using namespace Demo ;
              a=b=x=100; cout << a << " " << b << " " << x;  return 0; }
```

☛ Here the ***Demo*** namespace contains both ***a*** and ***b***, but not ***x***.

❒ ***Example 3*** (*Explicit **std :: qualification***)***:*** Standard C++ defines its entire library in its **own namespace**, ***std***. This is the reason that in our all previous programs we've used : ***using namespace std;***

☛ This causes the ***std namespace*** to be brought into the ***current namespace***, which gives you direct access to the ***names of the functions*** and ***classes*** defined within the library without having to qualify each one with ***std ::*** .

☛ We can explicitly qualify each name with ***std::*** . For example, following does not bring the ***standard library*** into the ***global namespace***.

```
#include <iostream>
int main(){   double val;
              std :: cout << " Enter a number : ";
              std :: cin >> val ;
              std :: cout << " This is your number : "<< val;
        return 0; }
```

➢ Here ***cout*** and ***cin*** are both ***explicitly qualified*** by their namespace. That is, to write a **standard output** you must specify ***std::cout***, and to read from **standard input** you must use ***std::cin***.

☛ However if you are using only a few names form the ***standard library***, you can still use those names ***without*** an ***std:: qualification*** but you will not be bringing the entire standard library into the global namespace. You have to declare the function names after ***#include <iostream>*** using the form: ***using std :: function_names;***

```
#include <iostream>
    /* gain access to cout and cin */
using std :: cout;
using std :: cin;
```

```
int main(){   double val;
              cout << " Enter a number : ";
              cin >> val ;
              cout << " This is your number: "<< val;
        return 0; }
```

➢ Here ***cin*** and ***cout*** can be used directly, but the rest of the ***std namespace*** has ***not been brought into view***.

❒ ***Example 4(Replace `static` with `namespace`):*** In C, if you want to **restrict the scope of a global name** to the file in which it is declared, you declare that name as ***`static`***. In C++ the use of ***`static` global declarations*** is still allowed, but a better way to accomplish the same result is to use an ***unnamed `namespace`.***

In C : Consider the following two files that are part of the same program:

File One	*File Two*

static int counter ;
void f1(){
counter = 99; /* OK */
}
	extern int counter ;
	void f2(){
	counter=10; /*error */
	}

- Since counter is defined in File **One**, it can be used in File **One**. In File **Two**, even though counter is specified as **extern**, it is still unavailable, & any attempt to use it results in an **error**.
- By preceding ***`counter`*** with ***`static`*** in File ***One***, the programmer has restricted its ***scope*** to that file.

In C++ : A better way to accomplish the same end is to use an ***unnamed namespace***

File One	*File Two*

/* unnamed namespace */
namespace { int counter ; }
void f1(){
counter = 99; /* OK */
}
	extern int counter ;
	void f2(){
	counter=10; /*error */
	}

- Here ***`counter`*** is also restricted to File **One**.
- The use of the ***unnamed namespace*** rather than ***`static`*** is the method ***recommended*** by **Standard C++.**

14.2 Conversion function (CvF)

To convert an object of one type into an object of another we can use an ***overloaded operator function***. However an ***easier (and better)*** way is : using a ***conversion function***. It allows an object to be included directly in an expression involving the **target type**.

❒ A conversion function converts an object into a value compatible with another type, which is often one of the built-in C++ types. In essence, ***a conversion function automatically converts an object into a value that is compatible with the type of the expression in which the object is used***. The general form of a conversion function:

`operator type(){ return value ; }`

- ➲ Here ***`type`*** is the **target type** you will be converting to
- ➲ ***`value`*** is the ***value of the object*** after the conversion has been performed.
- ➲ Conversion functions return a value of type ***`type`***.
- ➲ No parameters can be specified, and
- ➲ A conversion function ***must be a member of the class for which it performs the conversion***.

❒ ***Example 1:*** In the following program, the ***`coord`*** class contains a **conversion function** that converts an object to an ***`integer`***. In this case, the function returns the produce of the two coordinates;

```
class coord { int x, y;
        public :
            coord (int i, int j) { x = i; y = j; }
            operator int() { return x*y; }        /* conversion function */
        };
```

```
int main(){   coord o1(2, 3), o2(4, 3);
              int i;
              i = o1;      /* automatically convert to integer */
              cout << i << '\n';
              i = 100 + o2;          /* convert o2 to integer */
              cout << i << '\n';
          return 0;}
```

- ☛ Notice that the conversion function ***`operator int() { return x*y; }`*** is called when ***`o1`*** is assigned to an ***`integer`*** and when ***`o2`*** is used as part of a larger integer expression.
- ☛ By using a conversion function, you allow classes that you create to be integrated into "normal" C++ expressions without having to create a series of complex overloaded operator functions.

❒ ***Example 2:*** Following converts a ***`string`*** of type ***`strtype`*** into a **character pointer** to ***`str`***.

```
#include <iostream >
#include <cstring >
using namespace std;
class strtype { char str[80];
                int len ;
        public :
            strtype ( char *s) { strcpy (str , s);  len = strlen (s);  }
            operator char *() { return str; }        /* CvF : convert to char * */
           };
```

```
int main() { strtype  s(" This is a test ");
             char *p, s2[80];
             p = s;              /* convert to char *  */
             cout << " Here is string : " << p << '\n';
        /* convert to char * in function call */
             strcpy (s2, s);
             cout << " Here is copy of string : "
                  << s2 << '\n';
         return 0; }
```

- ☛ As you can see, not only is the conversion function invoked when ***object `s` is assigned to `p`*** (which is of type ***`char *`***), but it is also called when ***`s`*** is used as a **parameter** to ***`strcpy()`***.
- ☛ Conversion function also helps you to *integrate your classes into C++'s standard library functions*. We know ***`strcpy()`*** has the prototype:

`char *strcpy( char *s1, const char *s2);`

Because the prototype specifies that ***`s2`*** is of type ***`char *`*** , the conversion function to ***`char *`*** is automatically called.

14.3 `static` Class Members

class member variables can be declared as ***static***. By using ***static member variables***, you can bypass a number of rather tricky problems.

❒ When you declare a member variable as ***`static`***, you cause **only one copy of that variable to exist**-no matter how many **objects** of that **class** are created. Each **object** simply shares ***that one variable***.

- ☛ Also, the same ***static variable*** will be used by any **classes derived from** the class that contains the ***static member***.
  - ➢ Remember, ***for a normal member variable***, each time an object is created, a new copy of that variable is created, and that copy is accessible only by that object. While ***static variable*** is shared by all objects.

- ☛ A ***static member variable*** exists before any object of its class is created. In fact, ***it is actually possible to access a static member variable independent of any object***.
- ☛ In essence, a ***static class member*** is a ***global variable*** that simply has its ***scope restricted*** to the class in which it is declared.

❒ When you declare a ***static data member*** within a class, you must provide a ***definition for it elsewhere***, ***outside the class***. To do this, you ***re-declare*** the ***static variable***, using the ***scope resolution operator*** to identify which class it belongs to.

❒ ***Static member function:*** A ***member function*** can be declared as **`static`**, but it is ***not common***. A ***static member function*** can access only other ***static members*** of its class. Also it can access ***non-static global data*** and ***functions***.

- ☛ A ***static member function*** can be invoked by an object of ***its class***, or it can be called ***independent*** of any object, via the ***class name*** and the ***scope resolution operator***.
  - ○ A static member function does not have a ***this*** pointer.
  - ○ ***Virtual static*** member functions are ***not allowed***.
  - ○ Static member functions **cannot** be declared as ***const*** or ***volatile***.

NOTE

[1] All ***static member variables*** are ***initialized to 0*** by default. However, you can give a ***static class variable*** an initial value of your choosing.

[2] ***Static member variables*** helps to avoid the need for ***global variables***. Because classes that rely upon ***global variables*** almost always violate the ***encapsulation principle*** that is so fundamental to OOP and C++.

❒ ***Example 1:*** Following uses a ***static member variable***:

```
class myclass{     static int i;                    /* static private variable*/
      public :     void seti(int n){ i = n; }
                   int geti(){ return i; }
           };

int myclass :: i;        /* Definition of myclass :i. i is still private to myclass .*/
```

```
int main(){ myclass o1, o2;
            o1.seti(10) ;
cout << "o1.i: " << o1.geti() << '\n';        /* displays 10 */
cout << "o2.i: " << o2.geti() << '\n';   /* also displays 10 */
return 0; }
```

- ☛ Only object ***o1*** actually sets the value of **`static`** member ***i***. However, since ***i*** is shared by both ***o1*** and ***o2*** both calls to ***geti()*** display the same result.
- ☛ Notice, ***i*** is declared within ***myclass*** but defined outside of the class. Separate definition ensures that storage for ***i*** is defined.
- ☛ Technically, a ***class declaration*** is only a ***declaration***, no memory is actually set aside for that ***declaration***. Since a static data member implies that memory is allocated for that member, a separate definition is required that causes storage to be allocated.

❒ ***Example 2:*** **`static`** member can be accessed within a program ***independent of any object***. The following modified preceding program sets the value of ***i*** to ***100*** without any reference to a specific object.

```
class myclass{ public : static int i;      /* static changed topublic variable*/
                        void seti(int n){ i = n; }
                        int geti(){ return i; }
           };

int myclass :: i;        /* Definition of myclass :i. i is still private to myclass .*/
```

```
int main(){ myclass o1, o2;
            myclass::i = 100              /* set i directly */
cout << "o1.i: " << o1.geti() << '\n';        /* displays 10 */
cout << "o2.i: " << o2.geti() << '\n';   /* also displays 10 */
return 0; }
```

- ☛ Notice in "***myclass ::i = 100;***" the use of the ***scope resolution operator*** and ***class name*** to access ***i***. Here no object is referenced, ***i*** is set directly.

❒ ***Example 3: static member functions*** have limited applications, but one good use for them is to "***pre-initialize***" ***private static data*** before any object is actually created. Consider following,

```
class static_func_demo{  static int i;                        /* static member variable */
                public:  static void init(int x){i = x;}  /* static member function. It initializing i .*/
                         void show(){ cout << i; }
                      };

int static_func_demo ::i;                                     /* define i  */

int main(){static_func_demo :: init(100) ;          /* initializing static data before object creation */
           static_func_demo x;                                /* creating object x */
            x.show();                                         /* displays 100 */
        return 0;}
```

- ☛ Here ***i*** is initialized by the call to ***init()*** before an object of ***static_func_demo*** exists.

14.4 const MEMBER FUNCTIONS AND mutable

Class member functions can be declared as ***const***. When this is done, that function ***cannot modify the object that invokes it*** (Also, a ***const object*** cannot invoke a ***non-const member function***).

- ☛ A ***const member function*** can be called by either const or non-const objects.

❒ ***const:*** To specify a member function as ***const***, use the keyword "**`const`**" after/following the function's parameter declaration. Eg:

```
class X{ int some_var ;
   public :
         int f1() const ;          /* const member function */
      };
```

❒ ***mutable:*** To overrides the ***const-ness*** of a **`const`** function, use **`mutable`**. It gives ability to a **`const`** function to modify one or more selected members of a class (among the other members which we don't want to modify).

- ☛ That is, a ***mutable member*** can be modified by a ***const member function***.

❒ ***Example 1:*** The purpose of declaring a member function as **`const`** is to ***prevent it from modifying the object that invokes it***.

```
class Demo {      int i;
        public :  int geti() const { return i; }      /* ok */
                  void seti(int x) const { i = x; }   /* error! i can't be set from a const function */
          };
```

```
int main() { Demo ob;
             ob.seti(1900) ;
             cout << ob.geti();
           return 0; }
```

☛ This program ***will not compile*** because **`seti()`** is declared as **`const`**. This means that it is not allowed to modify the invoking object. Since it attempts to change **`i`**, the program is in error.

☛ Since **`geti()`** does not attempt to modify **`i`**, it is perfectly acceptable.

❒ ***Example 2:*** To allow selected members to be modified by a **`const`** ***member function***, specify them as **`mutable`**.

```
class Demo {  mutable int i;
              int j;
        public :
              int geti() const { return i; }      /* ok */
              void seti(int x) const {i = x;}     /* now, OK. Since i is mutable and can be modified
                                                     from const function */
//      following Still Wrong ! since j isn't mutable and const function can't modify it
//      void setj(int x) const { j = x; }
              } ;
```

```
int main() { Demo ob;
             ob.seti(1900) ;
             cout << ob.geti();
           return 0; }
```

☛ Here **`i`** is specified as mutable, so it can be changed by the **`seti()`** function.

☛ However, **`j`** is not mutable, so **`setj()`** is unable to modify its value.

14.5 Initializing object using "=" and the "explicit" specifier

Consider the following program:

```
class myclass{      int a;
            public:
                    myclass(int x){a=x;}      /* constructor */
                    int geta(){return a;}  };
```

```
int main(){  myclass ob(4) ;
             cout << ob.geta();
       return 0;}
```

◘ Here the ***constructor*** for **`myclass`** takes one parameter. Notice, how **`ob`** is declared in **`main()`**. The value **`4`**, specified in the parentheses following **`ob`**, is the argument that is passed to **`myclass()`**'s parameter **`x`**, which is used to initialize **`a`**.

☛ However, there is an ***alternative way to initialize objects***. For example, the following statement also initializes **`a`** to **`4`**:

```
myclass ob = 4;     /* automatically converts into myclass(4) */
```

➢ This form of initialization is automatically converted into a call to the **`myclass`** constructor with **`4`** as the argument. That is, the preceding statement is handled by the compiler as if it were written like this: **`myclass ob(4);`**

◘ In general, any time that you have a ***constructor that requires only one argument***, you can use either **`ob(x)`** or **`ob = x`** to initialize an object. The reason for this is that whenever you create a ***constructor that takes one argument***, you are also implicitly creating a ***conversion from the type of that argument to the type of the class***.

◘ ***The "explicit" specifier:*** If you do not want ***implicit conversions*** to be made, you can prevent them by using **`explicit`**. The ***explicit specifier*** applies only to ***constructors***. A constructor specified as explicit will be used only when an initialization uses the ***normal constructor syntax*** (i.e. **`ob(x)`**). It will not perform any **automatic conversion**. Here is **`myclass()`** declared as **`explicit`**:

```
explicit myclass(int x){ a = x; }
```

☛ Now only constructors of the form **`myclass ob(110);`** will be allowed.

❒ ***Example 1:*** There can be **more than one *converting constructor* in a class**. For example, consider this variation on **`myclass`**:

```
class myclass {    /*.... */
      public :  myclass (int x) { a = x; }
                myclass ( char *str ) { a = atoi (str ); }
      /*.... */};
```

```
int main() {myclass ob1 = 4;          /* converts to myclass(4) */
            myclass ob2 = "123 ";     /* converts to myclass("123") */
          /*.... */;}
```

☛ Since both constructors use different type arguments (as, of course, they must), each initialization statement ***is automatically converted into its equivalent constructor call***.

❒ ***Example 2(update via "="):*** The ***automatic conversion*** from the ***type of a constructor's first argument*** into a call to the ***constructor itself*** has interesting implications. Consider **`myclass`** from ***Example 1***, the following **`main()`** function makes use of the conversions from **`int`** and **`char *`** to assign ***ob1*** and ***ob2 new values***.

```
int main(){  myclass ob1 = 4;                  /* converts to myclass(4) */
             myclass ob2 = "123 ";             /* converts to myclass ("123") */

      /* use automatic conversions to assign new values */
             ob1 = "1776";                     /* converts into ob1 = myclass ("1776"); */
             ob1 = 2001;                       /* converts into ob1 = myclass (2001); */
           }
```

❒ ***Example 3:*** To ***prevent the conversions just shown***, you could specify the constructors as ***explicit***, as shown here:

```
#include <iostream >
#include <cstdlib >        /* use it for atoi() */
using namespace std;
class myclass {     int a;
          public:   explicit myclass(int x){ a = x; }              /* prevent auto conversion */
                    explicit myclass(char *str){ a = atoi(str); }  /* prevent auto conversion */
                    int geta(){ return a; }
            };
```

14.6 LINKAGE specifier for linking other language. asm for linking assembly language.

Linkage specifier tells the compiler that one or more functions in your C++ program will be linked with another language that might have a different approach to ***naming***, ***parameter passing***, ***stack restoration***, and the like.

- All C++ compilers allow functions to be linked as either ***C*** or ***C++*** functions. Some also allow to link functions with ***Pascal***, ***Ada***, Or ***FORTRAN***.
  - To cause ***a function to be linked*** for a different language, use this general form of the linkage specification:

    extern "language" function_prototype ;
    - Here ***language*** is the name of the language with which you want the specified function to ***link***.
  - If you want to specify linkage for ***more than one function***, use this form of the linkage specifcation:

    extern "language" { function_prototypes }
  - All ***linkage specification*** must be ***global***; they cannot be used inside a function.
  - The most common use of ***linkage specifications*** occur when linking ***C++*** programs ***to C*** code.
  - For example following links ***func()*** as a ***C***, rather than a ***C++***, function which can be compiled by a ***C compiler***:

```
extern "C" int func(int x);        /* link as C function */
```

  - The fragment tells the compiler that ***f1()***, ***f2()***, and ***f3()*** should be linked as ***C*** functions:

```
extern "C" { void f1();
             int f2(int x);
             double f3(double x, int *p); }
```

- ***asm:*** To link **assembly language routines** with a C++ program use the special keyword ***asm***, which allows you to embed ***assembly language instructions*** within a C++ function. These instructions are then compiled as is.
  - The advantage of using an ***in-line assembler*** is that your entire program is completely defined as a C++ program and there is no need to link ***separate assembly language files***. The general form of the ***asm*** keyword is shown here:

    asm("op_code");
    - where ***op_code*** is the assembly language instruction that will be embedded in your program.
  - This fragment embeds several ***assembly*** language instructions into ***func()***:

```
/* Don 't try this function! */
void func(){ asm("mov bp , sp");
             asm(" push ax");
             asm("mov c1 , 4");
           // ...}
```

NOTE:

[1] several compilers accept these three slightly different forms of the asm statement:

asm op_code; ***asm op_code newline*** ***asm { instruction sequence }***

- Here ***op-code*** is not enclosed in double quotes. Because embedded assembly language instruction tends to be implementation dependent, you will want to read your compiler's user manual on this issue.

[2] Microsoft Visual C++ uses _ _asm for embedding assembly code. It is otherwise similar to asm.

[3] You must be an accomplished assembly language programmer in order to successfully use in-line assembly language.

14.7 ARRAY-BASED I/O (Not will be used)

In addition to ***console*** and ***file I/O***, C++ supports a full set of functions that use ***character arrays*** as the ***input or output device***.

- ***Array-base I/O*** still operates through **streams**. All previous C++ **I/O operation** (including the ***binary I/O*** functions and the ***random-access*** functions)is applicable to **array-based I/O**. Before you can use ***array-based I/O***, you must be sure to include the header ***<strstream>*** in your file. In this header are defined the classes ***istrstream***, ***ostrstream***, and ***strstream***. These classes create array-based ***input***, ***output***, and ***input/output*** stream, respectively.
- ***ios*** is the **base** for these classes, so all the functions and manipulators included in ***istream***, ***ostream***, and ***iostream*** are also available in those ***array-based I/O classes***.
- To use a **character array** for ***output***, use this general form of the ***ostrstream*** constructor:

  ostrstream ostr(char *buf, streamsize size, openmode mode = ios :: out);
  - Here ***ostr*** will be the stream associated with the array ***buf***.
  - The ***size of the array*** is specified by ***size***.
  - ***mode*** is simply defaulted to output, but we can use any **output mode flag** defined by ***ios***. (***12.7*** for details.)
  - Once an ***array*** has been opened for ***output***, characters will be put into the array ***until it is full***. They array will not be ***overrun***. Any attempt to ***overfill*** the array will result in an ***I/O error***.
  - To find out how many characters have been written to the array, use the ***pcount()*** member function and must be called in ***conjunction*** with a ***stream***, and it will return the number of characters written to the array, including any ***null terminator***.

    streamsize pcount();
- To open an **array** for ***input***, use this form of the ***istrstream*** constructor: ***istrstream(const char *buf);***
  - ***buf*** is a **pointer** to the array that will be used for ***input***.
  - The input stream will be called ***istr***.
  - When ***input*** is being read from an array, ***eof()*** will return ***true*** when the ***end of the array*** has been reached.
- To open an **array** for ***input/output*** operations, use this form of the ***strstream*** constructor:

  strstream iostr(char *buf, streamsize size, openmode mode = ios::in | ios::out);
  - Here ***iostr*** will be an ***input output stream*** that uses the array **pointed** to by ***buf***, which is ***size*** characters long.

NOTE:

[1] The **character-based stream *classes*** are still valid, but future versions of the C++ language might not support them.

[2] The **character-based streams** are also referred to as ***char * streams*** in some C++ literature.

❒ ***Example 1:*** Here is a short example that shows how to open an array for output and write data to it:

```
#include <iostream>
#include <strstream>
using namespace std;
int main(){char buf[255];              /* output buffer */
        ostrstream ostr(buf , sizeof buf);          /* open output array */
        ostr << "Array - based I/O uses streams just like ";
        ostr << "'normal ' I/O\n" << 100;
        ostr << ' ' << 123.23 << '\n';
        /* you can use manipulators , too */
        ostr << hex << 100 << ' ';
        /* or format flags */
        ostr . setf (ios :: scientific );
        ostr << dec << 123.23;
        ostr << endl << ends ;
        cout << buf ; /* show resultant string */
return 0; }
```

☛ This program **manually null-terminates** the output array by using the ***ends*** **manipulator**. Whether the array will be automatically null terminated or not is ***implementation dependent***, so it is best to perform this ***manually if null termination is important to your application***.

❒ ***Example 2:*** Here is an example of array-based input:

```
int main(){  char buf[255] = " Hello 100 123.125 a";
             istrstream istr(buf);            /* open input array */
             int i;                    /* for integers in buf */
             char str[80];             /* for strings in buf */
             float f;                  /* for floates in buf */
             char c;                   /* for charecters in buf */
             istr >> str >> i >> f >> c;          /* read all strings, integers, floates and charecters in buf */
             cout<< str<<' '<< i <<' '<< f <<' '<< c; /* display all strings, integers, floats and charecters in buf */
     return 0; }
```

☛ The program ***reads*** and then ***redisplays*** the values contained in the input array ***buf***.

❒ ***Example 3:*** Keep in mind that an input array, once linked to a stream, will appear the same as a file. Following uses ***get()*** and the ***eof()*** function to read the contents of ***buf*** of previous example:

```
while( !istr.eof() ){ istr.get(c);
                      if( !istr.eof() ) cout << c;
```

❒ ***Example 4:*** Following performs input and output on an array:

```
int main(){  char iobuf[255];
             strstream iostr(iobuf, sizeof iobuf);

             iostr << " This is a test \n";
             iostr << 100 << hex <<' '<< 100<< ends;

             char str[80];
             int i;
```

```
iostr.getline(str, 79); /* read string up to '\n' */
iostr >> dec >> i;          /* read 100 */
cout << str << ' ' << i << ' ';
iostr >> hex >> i;
cout << hex << i;
return 0; }
```

➲ The program first writes output to ***iobuf***. It then reads it back.

➲ It first reads the entire line "***This is a test***" using the ***getline()*** function. It then reads the decimal value ***100*** and the hexadecimal value ***0x64***.

Standard Template Library (STL)

STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. The ***STL*** is a large library, and not all of its features can be described in this chapter. The overview presented here is intended to familiarize you with its basic operation, design philosophy, and programming fundamentals. Sometimes ***STL*** may look more complicated than it actually is.

❒ The ***STL*** provides general-purpose, ***templatized classes*** and ***functions*** that implement many popular and commonly used ***algorithms and data structures***.

➲ For example, it includes support for ***vectors***, ***lists***, ***queues***, and ***stacks***. It also defines various routines that access them.

➲ Because the ***STL*** is constructed from ***template classes***, the ***algorithms*** and ***data structures*** can be applied to ***any type of data***.

14.8 An Overview Of STL

There are three main items at the core of the ***STL***, these are: ***containers***, ***algorithms***, and ***iterators***. These items work in conjunction with one another to provide off-the-shelf solutions to a variety of programming problems.

❒ ***Containers:*** **Containers** are objects that hold other objects. Containers are the ***STL objects*** that actually store data. There are several different types of containers. The containers defined by the ***STL*** are shown in following **TABLE**. Also shown are the ***headers*** you must include ***to use each container***.

Container	***Description***	***Required Header***
bitset	A set of bits	**<bitset>**
deque	A double-ended queue	**<deque>**
list	A linear list	**<list>**
map	Stores key/value pairs in which each key is associated with only one value	**<map>**
multimap	Stores key/value pairs in which one key can be associated with two or more values	**<map>**
multiset	A set in which each element is not necessarily unique	**<set>**
priority_queue	A priority queue	**<queue>**
queue	A queue	**<queue>**
set	A set in which each element is unique	**<set>**
stack	A stack	**<stack>**
vector	A dynamic array	**<vector>**

- ➲ The ***string*** class, which ***manages character strings***, is also a ***container***, but it is discussed later in this chapter.
- ➲ Each ***container class*** defines a **set of functions** that can be applied to the ***container***. For example,
  - ▶ A ***list*** container includes functions that ***insert***, ***delete***, and ***merge*** elements.
  - ▶ A ***stack*** container includes functions that ***push*** and ***pop*** values.
- ➲ ***Associative containers:*** In addition to the **basic containers**, the ***STL*** also defines ***associative containers***, which allow ***efficient retrieval of values based on keys***. For example, the ***map*** class defines a map that provides ***access to values with unique keys***. Thus, a ***map*** stores a ***key/value pair*** and allows a value to be retrieved when its key is given.

❒ ***Algorithms:*** ***Algorithms*** act on ***containers***. Some of the **services algorithms** perform are ***initializing***, ***sorting***, ***searching***, and ***transforming the contents of containers***. Some operate on a ***sequence***, which is a ***linear list*** of elements within a container.

❒ ***Iterators:*** ***Iterators*** are ***objects*** that are, more or less, ***pointers***. They used for ***cycle through the contents of a container*** in much the same way that a ***pointer*** used for ***cycle through an array***. ***Iterators*** are declared using the ***iterator type*** defined by the ***various containers***. The five types of ***iterators*** are:

Iterator	*Access Allowed*
Random access	***Stores*** and ***retrieves*** values. Elements can be accessed ***randomly***
Bidirectional	***Stores*** and ***retrieves*** values. ***Forward*** and ***backward*** moving.
Forward	***Stores*** and ***retrieves*** values. ***Forward*** moving only.
Input	***Retrieves*** but does not store values. ***Forward*** moving only
Output	***Stores*** but does not retrieve values. ***Forward*** moving only

- ⇨ In general, an ***iterator*** that has ***greater access capabilities*** can be used in place of one that has ***lesser capabilities***. For example, a ***forward*** iterator can be used in place of an ***input*** iterator.
- ⇨ ***Iterators*** are handled just like ***pointers***. We can ***increment(++)*** and ***decrement(--)*** them. We can apply the ***** operator.
- ⇨ ***reverse iterators:*** ***Reverse iterators*** are either ***bidirectional*** or ***random-access*** iterators that move through a sequence in **reverse direction**. Thus, if a ***reverse iterator*** points to the ***end of a sequence***, ***incrementing*** that iterator will cause it to point to ***one element before the end***.
- ⇨ Following iterator types will be used later in this chapter.

Term	BiIter	ForIter	InIter	OutIter	RandIter
Iterator Type	Bidirectional iterator	Forward iterator	Input iterator	Output iterator	Random-access iterator

❒ ***STL*** also relies upon several other standard components for support. Important among these are ***allocators***, ***predicates***, and ***comparison functions***.

- ❖ ***Allocator :*** Each ***container*** has an ***allocator*** defined for it. **Allocators** manage ***memory allocation*** for **containers**. The ***default allocator*** is an **object of class** ***allocator***, but you can define your ***own allocators for specialized applications***. For most uses, the ***default allocator*** is sufficient.
- ❖ ***Predicate:*** Some ***algorithms*** and ***containers*** use a ***special type of function*** called a ***predicate***. There are two type of ***predicates***:
  [1] ***unary:*** A unary predicate takes one argument [2] ***binary:*** a binary predicate has two arguments.
  - These functions return ***true/false***; the ***conditions*** that make them return ***true/false*** are defined by the programmer.
  - ***Type notation:*** In this chapter, when a ***unary predicate function*** is used, it will be notated with the type ***UnPred***. When a ***binary predicate*** is used, it will be of type ***BinPred***.
  - In a binary predicate, the arguments are always in the order of first, second.
  - For both ***unary*** and ***binary***, the ***arguments*** will be the ***same type as the objects*** being stored by the ***container***.
- ❖ ***Comparison function:*** ***Comparison function*** is a special type of ***binary predicate*** that compares two elements, this type of predicate returns ***true*** if its ***first argument is less than its second***.
  - ▶ ***Comparison functions*** are used by some ***algorithms*** and ***classes*** and it will be notated by the **type** ***Comp***.

❒ The C++ ***STL*** includes the ***<utility>*** and ***<functional>*** headers, which ***provide support for the STL***. For example, ***<utility>*** contains the definition of the ***template class pair***, which can hold a pair of values.

- ❖ Templates in ***<functional>*** help to construct objects that define ***operator()***. These are called ***function objects***, and they can be used in place of ***function pointer*** in many places. Some predefined ***function objects*** declared within ***<functional>*** are:

plus	*minus*	*multiplies*	*divides*	*modulus*	*negate*	*equal_to*	*not_equal_to*
greater	*greater_equal*	*less*	*less_equal*	*logical_and*	*logical_or*	*logical_not*	

  - ▶ Most widely used function object is ***less***, which determines whether the value of one object is less than the value of another.
- ❖ ***Function objects*** can be used in place of actual ***function pointers*** in the STL algorithms. Using ***function objects*** rather than ***function pointers*** allows the ***STL*** to generate more ***efficient code***.

★ NOTE: Function objects are not difficult but we for now we have to skip this.

14.9 Type Names (Placeholder Types) For Container Classes

Because the names of the ***placeholder types*** in a **template class declaration** are arbitrary, the container classes declare ***typedef***ed versions of these types (***typedef*** is a C specifier. Recall ***typedef 7.6***). This makes the ***type names*** concrete. Common ***typedef*** names :

typedef	*Name Description*	*typedef*	*Name Description*
`reference`	A ***reference*** to an element	`size_type`	An ***integral type*** equivalent to ***size t***
`iterator`	An ***iterator***	`const_reference`	A ***const reference*** to an element
`const_iterator`	A ***const*** iterator	`value_type`	The ***type*** of a ***value*** stored in a container
`reverse_iterator`	A ***reverse*** iterator	`key_compare`	The type of a function that ***compares*** two ***keys***
`const_reverse_iterator`	A ***const reverse*** iterator	`key_type`	The ***type*** of a ***key***
`allocator_type`	The ***type*** of the allocator	`value_compare`	The type of a function that ***compares*** two ***values***

14.10 VECTORS

Limitations of an array and purpose of "vector": The main limitation of C++ array is that the ***size of an array*** is fixed at compile time and cannot be adjusted at run time to accommodate ***changing program conditions***. A ***vector*** solves this problem by ***allocating memory*** as needed.

- The ***vector*** class supports a ***dynamic array***. This is an array that ***can grow as needed***. ***Vector*** also supports ***Random Access***.
  - Although a ***vector*** is ***dynamic***, the subscripting operator **[ ]** is defined for ***vector*** and you can still use the ***standard array subscript notation*** to access its elements.
  - These ***comparison operators*** are defined for vector: **==, <, <=, !=, >, >=**
- ***The* `template` *specification for* `vector`*:***

```
template <class T, class Allocator = allocator<T>> class vector
```

  - Here **T** is the ***type*** of data being stored
  - ***Allocator*** specifies the allocator, which defaults to the **standard allocator**
- ***vector*** has the following ***constructors*** :

[1] This form constructs an ***empty vector :***

```
explicit vector(const Allocator &a=Allocator() );
```

[2] The second form constructs a ***vector*** that has ***num*** elements with the value ***val***. The value of ***val*** can be allowed to default :

```
explicit vector(size_type num, const T &val=T(), const Allocator &a=Allocator() );
```

[3] The third form constructs a ***vector*** that contains the **same elements** as ***ob***.

```
vector( const vector<T, Allocator> &ob);
```

[4] The fourth form constructs a ***vector*** that contains the **elements in the range** specified by the ***iterators start*** and ***end***. (This constructor uses **specific instance**):

```
template<class InIter> vector(InIter start, InIter end, const Allocator &a=Allocator());
```

- Any **object-type** that will be stored in a ***vector*** must define a ***default constructor***. It must also define the **<** and **==** operations.
- Some examples about declaring a ***vector :***

```
vector <int> iv;           /* creates a zero - length int vector */
vector <char> cv(5);       /* creates a 5- element char vector */
vector <char> cv(5, 'x');  /* initializes a 5- element char vector */
vector <int> iv2(iv);      /* creates an int vector from an int vector */
```

- The ***member functions*** defined by ***vector*** are shown in following Table.

Member Function *(template portion in different line)*		***Description***
reference *at*(**size_type** i);	**const_reference** *at*(**size_type** i) **const**;	Returns a ***reference*** to an element specified by ***i***.
reference *back*();	**const_reference** *back*() **const**;	Returns a ***reference*** to the ***last*** element in the vector.
reference *front*();	**const_reference** *front*() **const**;	Returns a ***reference*** to the ***first*** element in the vector.
reference *operator*[ ](**size_type** i) **const**; **const_reference** *operator*[ ](**size_type** i) **const**;		Returns a ***reference*** to the element specified by ***i***.
iterator *begin*();	**const_iterator** *begin*() **const**;	Returns an ***iterator*** to the ***first*** element in the vector.
iterator *end*();	**const_iterator** *end*() **const**;	Returns an ***iterator*** to the ***end*** of the vector.
reverse_iterator *rbegin*();	**const_reverse_iterator** *rbegin*() **const**;	Returns a ***reverse iterator*** to the ***end*** of the vector.
reverse_iterator *rend*();	**const_reverse_iterator** *rend*() **const**;	Returns a ***reverse iterator*** to the ***start*** of the vector.
template<class InIter> **void** *assign*(**InIter** ***start***, **InIter** ***end***);		Assigns the vector the sequence defined by ***start*** and ***end***.
template<class Size, class T> **void** *assign*(Size ***num***, const T &***val***=T());		Assigns the vector ***num*** elements of value ***val***.
template<class InIter> **void** *insert*(iterator i, **InIter** ***start***, **InIter** ***end***);		Inserts the sequence defined by ***start*** and ***end*** immediately before the element specified by ***i***.
void *insert*(iterator i, **size_type** ***num***, const T &***val***)		Inserts ***num*** copies of ***val*** immediately ***before*** the element specified by ***i***.
iterator *insert*(iterator i, const T &val=T());		Inserts ***val*** immediately ***before*** element specified by ***i***. An iterator to element is returned.
iterator *erase*(iterator ***start***, iterator ***end***);		Removes elements in range ***start*** to ***end***. Returns an ***iterator*** to element ***after*** last element removed.
iterator *erase*(iterator i);		Removes element pointed to by ***i***. Returns an ***iterator*** to element ***after*** the one removed.
void *clear*();		***Removes*** all ***elements*** from the vector.
bool *empty*() **const**;		Returns ***true*** if the invoking vector is ***empty*** and ***false*** otherwise.
void *pop_back*();		***Removes*** the ***last*** element in the vector.
void *push_back*(**const** T &***val***);		***Adds*** an ***element*** with the value specified by ***val*** to the ***end*** of the vector.
void *resize*(**size_type**, ***num***, T ***val***=T());		***Changes*** the ***size*** of the vector to that specified by ***num***. If the vector must be lengthened, elements with the value specified by ***val*** are ***added*** to the ***end***.
size_type *capacity*() **const**;		Returns the ***current capacity*** of the vector. This is the number of elements it can hold before it will need to allocate more memory.
size_type *max_size*() **const**;		Returns the ***maximum number of elements*** that the vector can hold.
size_type *size*() **const**;		Returns the ***number of elements*** currently in the vector.
allocator_type *get_allocator*() **const**;		Returns the vector's ***allocator***.
void *reserve*(**size_type** ***num***);		Sets the capacity of the vector so that it is equal to at ***least num***.
void *swap*(vector<T, Allocator> &***ob***)		***Exchanges*** the elements stored in the **invoking vector** with those in ***ob***.

- Some of the most important member functions are:
  - [1] ***`size()`:*** It returns the ***current size*** of the vector. This function is quite useful because it allows you to determine the ***size*** of a vector at ***run time***. Remember, vectors will increase in size as needed, so the size of a vector must be determined during ***execution***, not during ***compilation***.
  - [2] ***`begin()`:*** The ***`begin()`*** function returns an ***iterator*** to the ***start*** of the vector.
  - [3] ***`end()`:*** The ***`end()`*** function returns an ***iterator*** to the ***end*** of the vector.

[4] ***push_back():*** The ***push_back()*** function puts a value onto the ***end*** of the vector. If necessary, the vector is ***increased*** in length to ***accommodate*** the ***new*** element.
[5] ***insert():*** You can also ***add*** elements to the ***middle*** using ***insert()***.
[6] ***erase():*** You can ***remove*** elements from a vector using ***erase()***.

NOTE:
[1] As explained, ***iterators*** are similar to ***pointers***, and it is through the use of the ***begin()*** and ***end()*** functions that you obtain an iterator to the beginning and end of a vector.
[2] A vector can also be initialized. In any event, once a vector contains elements, you can use ***array subscripting*** to access or modify those elements.

❒ ***Example 1:*** Folllowing illustrates the basic operation of a vector.

```
# include <iostream >
# include <vector >
using namespace std;
int main(){  vector <int> v;                    /* create zero - length vector */
             int i;

             cout << " Size = " << v.size() << endl ;       /* display original size of v */

/* put values onto end of vector vector will grow as needed */
             for(i=0; i<10; i++) v.push_back(i);
             cout << " Size now = " << v.size() << endl ;   /* display current size of v */

             cout << " Current contents :\n";
             for(i=0; i<v.size(); i++) cout<< v[i] << " ";  /* display contents of vector */
             cout << endl ;

/* put more values onto end of vector again , vector will grow as needed */
             for(i=0; i <10; i++) v.push_back(i+10) ;
             cout << " Size now = " << v.size() << endl;    /* display current size of v */
             cout << " Current contents :\n";
             for(i=0; i<v.size(); i++) cout<< v[i] << " ";  /* display contents of vector */
             cout << endl ;

             for(i=0; i<v.size(); i++) v[i] = v[i] + v[i];  /* change contents of vector */
             cout << " Current contents :\n";
             for(i=0; i<v.size(); i++) cout<< v[i] << " ";  /* display contents of vector */
             cout << endl ;

       return 0;}
```

OUTPUT:

```
Size = 0
Size now = 10
Current contents:
0 1 2 3 4 5 6 7 8 9
Size now = 20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Current contents:
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34
```

☞ In ***main()***, an integer vector ***v*** is created. Since no initialization is used, it is an empty vector with an initial capacity of zero.

☞ The program confirms it's a ***zero-length vector*** by calling the ***size()*** member function.

☞ Next, ***ten*** elements are added to the ***end*** of ***v*** with the member function ***push_back()***. This causes ***v*** to grown in order to accommodate the new elements. Its new size after these additions is ***10***. Then, the contents of ***v*** are displayed. Notice that the ***standard array subscripting notation*** is employed.

☞ Next, ***ten*** more elements are added and ***v automatically*** increased in size to handle them.

☞ Finally, the values of ***v***'s elements are ***altered*** using ***standard subscripting notation***.

☞ NOTICE that the ***loops that display the contents*** of ***v*** use as their target ***v.size()***.

☞ One of the ***advantages that vectors have over arrays*** is that it is possible to find the ***current size*** of a ***vector***.

❒ ***Example 2:*** We know that ***arrays*** and ***pointers*** are tightly linked in C++ also ***array*** can be accessed either ***through subscripting*** or ***through a pointer***. Similarly in ***STL vectors*** (in place of ***array***) and ***iterators*** (in place of ***pointer***) are linked and we can access the members of a ***vector*** by ***using subscripting*** or by ***using an iterator***.

```
int main(){ vector <int> v;                 /* create zero - length vector */
            int i;
    for(i=0; i<10; i++) v.push_back(i);     /* push values into a vector */
    for(i=0; i<10; i++) cout << v[i] << " "; /* accessing vector via subscript */
            cout << endl ;
```

```
                                            /* access via iterator */
vector <int> :: iterator p;                 /* iterator declaration */
p = v.begin();                              /* initializing iterator */
while (p != v.end()) { cout << *p << " ";
                       p++; }
return 0; }
```

☞ **v** is initially created with **zero length**. ***push_back()*** puts values onto the ***end*** of the vector, expanding its size as needed.

☞ Notice how the ***iterator p*** is declared. The ***type*** iterator is defined by the ***container*** classes. Thus, to obtain ***an iterator*** for a ***particular container***. simply **qualify** ***iterator*** with the ***name*** of the container.

☞ ***p*** is initialized to point to the start of the vector by using the ***begin()***. ***begin()*** returns an ***iterator*** to the ***start*** of the ***vector***.

☞ This ***iterator*** can then be used to access the vector an element at a time by ***incrementing*** it as needed. Which is similar to the way that ***a pointer can be used to access the elements of an array***.

☞ To determine the end of the vector, the ***end()*** is employed. ***end()*** returns an ***iterator*** to the location that is ***one past the last element in the vector***. Thus, when ***p*** equals ***v.end()***, the **end of the vector** has been **reached**.

❒ ***Example 3:*** you can insert elements into the middle using the ***insert()*** function. You can also remove elements using ***erase()***.

```
int main() { vector <int> v(5, 1); /* create 5-element vector of 1's */
int i;

cout << "Size = " << v.size() << endl ;
cout << " Original contents :\n";
for (i=0; i<v.size(); i++) cout << v[i] << " ";          /* display original contents */
cout << endl << endl ;

vector <int> :: iterator p = v.begin(); /* declaring iterator p and initializing it */
p += 2;                                 /* point to 3rd element */
v.insert(p, 10, 9);                     /* insert 10 elements with value 9 */
```

OUTPUT:

```
Size = 5                    Contents after insert:
Original contents:          1 1 9 9 9 9 9 9 9 9 9 1 1 1
1 1 1 1 1                   Size after erase = 5
Size after insert = 15      Contents after erase:
                            1 1 1 1 1
```

```
cout << "Size after insert =" << v.size() << endl ;
cout << "Contents after insert :\n";
for (i=0; i<v.size(); i++)
      cout<< v[i] << " "; /* show changed contents */
cout << endl << endl

          /* remove those elements */
p = v.begin();
p += 2;                     /* point to 3rd element */
v.erase(p, p +10) ;     /* remove next 10 elements */
cout << " Size after erase = " << v.size() << endl ;
cout << " Contents after erase :\n";
for (i=0; i<v.size(); i++)
      cout << v[i] << " ";      /* show final contents */
cout << endl ;
return 0;}
```

❒ ***Example 4:*** ***comparison operators need to be defined for user defined classes:*** Here is an example that uses a vector to store objects of a **programmer-defined class**.

☞ Notice that the class defines the ***default constructor*** and that ***overloaded versions*** of **<** an **==** are provided.

☞ **NOTE:** depending upon how your compiler implements the ***STL***, ***other comparison operators might need to be defined***.

```
class Demo{ double d;
      public :
          Demo() { d = 0.0; }
          Demo(double x) { d = x; }

/* advanced "=" overloading. recall 11.13*/
          Demo &operator=(double x) { d = x; return *this; }
          double getd() { return d; }
      };

bool operator<( Demo a, Demo b){ return a.getd() < b.getd(); }
bool operator==( Demo a, Demo b){ return a.getd() == b.getd(); }
```

```
int main(){ vector <Demo> v;
          int i;
          for(i=0; i<10; i++)v.push_back( Demo (i /3.0) );
          for(i=0; i<v.size(); i++) cout << v[i].getd() << " ";
          cout << endl ;

          for(i=0; i<v.size(); i++) v[i] = v[i].getd()*2.1;
          for(i=0; i<v.size(); i++) cout << v[i].getd() << " ";
return 0;}
```

OUTPUT:

```
0 0.333333 0.666667 1 1.33333 1.66667 2 2.33333 2.66667 3
0 0.7 1.4 2.1 2.8 3.5 4.2 4.9 5.6 6.3
```

14.11 LISTS

The ***list*** class supports a ***bidirectional***, ***linear list***. A ***list*** can be accessed ***sequentially only*** (but a ***vector*** supports ***random access***). Because lists are ***bidirectional***, they can be accessed ***front to back*** or ***back to front***.

❒ **`template` *specification for* `list`*:***

```
template < class T, class Allocator = allocator<T> > class list
```

▶ Here **T** is the ***type*** of data being stored ▶ ***Allocator*** specifies the allocator, which defaults to the **standard allocator**

▶ These ***comparison operators*** are defined for ***list***: **==, <, <=, !=, >, >=**

❒ ***list*** has the following ***constructors*** :

[1] This form constructs an ***empty list:***

```
explicit list(const Allocator &a=Allocator() );
```

[2] The second form constructs a ***list*** that has ***num*** elements with the value ***val***. The value of ***val*** can be allowed to default :

```
explicit list(size_type num, const T &val=T(), const Allocator &a=Allocator() );
```

[3] The third form constructs a ***list*** that contains the **same elements** as ***ob***.

```
list( const list<T, Allocator> &ob);
```

[4] The fourth form constructs a ***list*** that contains the **elements in the range** specified by the ***iterators*** ***start*** and ***end***. (This constructor uses **specific instance**):

```
template<class InIter> list(InIter start, InIter end, const Allocator &a=Allocator());
```

❒ Any **object-type** that will be held in a ***list*** must define a ***default constructor***. It must also define the comparison operators **==**, **<** etc.

❒ The ***member functions*** defined by ***list*** are shown in following Table.

Member Function		***Description***
reference *front*();	**const_reference** *front*() **const**;	Returns a ***reference*** to the ***first*** element in the list.
reference *back*();	**const_reference** *back*() **const**;	Returns a ***reference*** to the ***last*** element in the list.
iterator *begin*();	**const_iterator** *begin*() **const**;	Returns an ***iterator*** to the ***first*** element in the list.
iterator *end*();	**const_iterator** *end*() **const**;	Returns an ***iterator*** to the ***end*** of the list.
reverse_iterator *rbegin*();	**const_reverse_iterator** *rbegin*() **const**;	Returns a ***reverse iterator*** to the ***end*** of the list.
reverse_iterator *rend*();	**const_reverse_iterator** *rend*() **const**;	Returns a ***reverse iterator*** to the ***start*** of the list.
void *sort*();	**template<class** Comp>	***Sorts*** the list. The second for sorts the list using the comparison function ***cmpfn***

void *sort*(Comp ***cmpfn***);	to determine whether the value of one element is ***less*** than that of another.
void *unique*(); **template<class** BinPred> **void** *unique*(BinPred ***pr***);	***Removes duplicate*** elements from the invoking list. The second from uses ***pr*** to determine ***uniqueness***.
void *merge*(list<T,Allocator>&ob); **template<class** Comp> **void** *merge*(<list<T,Allocator>> &***ob***, Comp ***cmpfn***);	***Merges*** the ordered list contained in ***ob*** with the invoking ordered list. The result is ordered. After the merge, the list contained in ***ob*** is empty. In the second form, a comparison function can be specified to determine whether the value of one element is ***less*** than that of another.

void *splice*(**iterator** i, list<T, **Allocator**> &ob);	Inserts the contents of ***ob*** into the invoking list at the location pointed to by ***i***. After the operation ***ob*** is ***empty***.
void *splice*(**iterator** i, list<T, **Allocator**>&ob, **iterator** el);	Removes the element pointed to by ***el*** from the list ***ob*** and stores it in the invoking list at the location pointed to by ***i***.
void *splice*(**iterator** i, list<T, **Allocator**> &ob, **iterator** start, **iterator** end);	Removes the range defined by ***start*** and ***end*** from ***ob*** and stores it in the invoking list beginning at the location pointed by ***i***.

template<class InIter> void *assign*(**InIter** ***start***, **InIter** ***end***);	Assigns the list the sequence defined by ***start*** and ***end***.
template<class Size, **class** T> **void** *assign*(**Size** ***num***, **const** T &***val***=T());	Assigns the list ***num*** elements of value ***val***.
void *remove*(**const** T &val);	***Remove elements*** with the value ***val*** from the list.
template<class UnPred> **void** *remove_*if(UnPred ***pr***);	***Removes elements*** for which the unary predicate ***pr*** is ***true***.
template<class InIter> void *insert*(iterator i, **InIter** *start*, **InIter** *end*);	Inserts the sequence defined by ***start*** and ***end*** immediately ***before*** the element specified by ***i***.

iterator *insert*(**iterator** i, **const** T &***val***=T());	Inserts ***val*** immediately ***before*** the element specified by ***i***. Iterator returned to the element
void *insert*(**iterator** *i*, **size_type** num, **const** T &***val***)	Inserts ***num*** copies of ***val*** immediately ***before*** the element specified by ***i***.
allocator_type *get_allocator*() **const**;	Returns the list's ***allocator***.
void *swap*(**list**<T, **Allocator**> &ob)	***Exchanges*** the elements stored in the invoking list with those in ***ob***.
iterator *erase*(**iterator** start, **iterator** end);	***Removes*** the ***elements*** in the range ***start*** to ***end***. Returns iterator to the element after last removed element.

iterator *erase*(**iterator** i);	***Removes element*** pointed to by ***i***. Returns iterator to the element after the one removed.
void *clear*();	***Removes all elements*** from the list.
bool *empty*() **const**;	Returns ***true*** if the invoking list is ***empty*** and ***false otherwise***.
void *pop_back*();	***Removes*** the ***last*** element in the list.
void *pop_front*();	***Removes*** the ***first*** element in the list.
void *push_back*(**const** T &***val***);	***Adds*** an ***element*** with the value specified by ***val*** to the ***end*** of the list.
void *push_front*(**const** T &***val***);	***Adds*** an ***element*** with the value specified by ***val*** to the ***front*** of the list.
void *resize*(**size_type**, num, T val=T());	***Changes*** the ***size*** of the list to that specified by ***num***. If the list must be lengthened, elements with the value specified by ***val*** are added to the ***end***.
size_type *max_size*() **const**;	Returns the ***maximum*** number of ***elements*** that the list can ***hold***.
size_type *size*() **const**;	Returns the ***number of elements*** currently in the list.
void reverse();	***Reverses*** the invoking list.

- Like a vector, a list can have elements put **into** it with the ***push_back()*** function.
- You can put elements **on the front** of the list by using ***push_front()***,
- you can insert an element **into the middle** of a list by using ***insert()***.
- You can use ***splice()*** to **join two lists**, and you can **merge one list** into another by using ***merge()***.

❒ ***Example 1:*** Following creates a list of characters.

```
# include <iostream >
# include <list >
using namespace std;
int main() { list <char> lst ;                          /* create an empty list */
             int i;
             for(i=0; i<10; i++) lst.push_back('A'+i);      /* creating list */
             cout << " Size = " << lst.size() << endl;      /* Displaying size */
             list <char >:: iterator p;          /* "pointer-like" iterator declaration */
             cout << " Contents : ";
             while(!lst.empty()){p = lst.begin();
                                 cout << *p;
                                 lst.pop_front();          /* removing printed element */
                                 }                          /* list output loop ends */
return 0; }
```

⇨ First, an empty list object is created. Next, ten characters, the letters ***A*** through ***J***, are put into the list. This is accomplished with the ***push_back()*** function, which puts each new value on the end of the existing list.

⇨ Next, the ***size*** of the list is displayed.

⇨ Then, the contents of the list are output by repeatedly ***obtaining***, ***displaying***, and then ***removing*** the **first element** in the list. This process stops when the list is **empty**.

⇨ Here the list was **emptied** as it was ***traversed***. That is, of course, not necessary. For example, the loop that displays the list could be **RE-CODED** as:

```
list <char> :: iterator p = lst.begin();
while(p != lst.end()){     cout << *p;   p++; }
```

▶ Here the ***iterator*** **p** is initialized to point to the ***start*** of the list. Each time through the loop, ***p*** is ***incremented***, causing it to point to the ***next element***. The loop ends when ***p*** points to the ***end*** of the list.

❒ ***Example2:*** Because lists are ***bidirectional***, elements can be put on a list ***either*** at the ***front*** or at the ***back***. The following program creates two list, with the first being the ***reverse*** of the second.

```
int main(){ list <char> lst ;
            list <char > revlst ;
            int i;
            for(i=0; i <10; i++) lst.push_back('A'+i);

            cout << " Size of lst = " << lst.size() << endl ;

            cout << " Original contents : ";
            list <char >:: iterator p;
    /* Remove elements from lst and put them into revlst in reverse order. */
            while(!lst.empty()) { p = lst.begin();
            cout << *p;
            lst.pop_front ();
            revlst.push_front(*p);      /* putting first as last : reversing */ }

cout << endl << endl ;

cout << " Size revlst = " << revlst.size() << endl ;
cout << " Reversed contents : ";
p = revlst.begin();
while (p!=revlst.end()){cout << *p;
                        p++;}
return 0;}
```

OUTPUT:
```
Size of lst = 10
Original contents: ABCDEFGHIJ
Size revlst = 10
Reversed contents: JIHGFEDCBA
```

☞ the list is reversed by removing elements from the start of ***`lst`*** and pushing them onto the front of ***`revlst`***. (stored in reverse order in ***revlst***)

❒ ***Example 3:*** sort a list by calling the ***`sort()`***member function.

```
#include <cstdlib >
lst.push_back('A'+( rand()%26) );      /* makes random charcter list */
list<char >:: iterator p;              /* iterator or pointer */
lst.sort();                            /* sorts the list */
```

OUTPUT
```
Original contents: PHQGHUMEAY
Sorted contents: AEGHHMPQUY
```

❒ ***Example 4:*** One ordered list can be ***merged*** with another. The result is an **ordered** list that contains the contents of the two **original** lists. The ***new list*** is left in the ***invoking list*** and the ***second list*** is left ***empty***.

```
int main(){ list <char> lst1 , lst2 ;
            int i;
            for(i=0; i <10; i +=2) lst1.push_back('A'+i);
            for(i=1; i <11; i +=2) lst2.push_back('A'+i);
            list <char> :: iterator p ;

cout << "Contents of lst1:";
p = lst1.begin();
while(p != lst1.end()){ cout << *p;
                        p++; }
cout << endl << endl ;

cout << "Contents of lst2:";
p = lst2.begin();
while(p != lst2.end()){cout << *p;
                       p++;}
cout << endl << endl ;
```

OUTPUT:
```
Contents of lst1: ACEGI
Contents of lst2: BDFHJ
lst2 is now empty
Contents of lst1 after merge:
ABCDEFGHIJ
```

```
lst1.merge( lst2 );
if( lst2.empty() ) cout << " lst2 is now empty \n";

cout << " Contents of lst1 after merge :\n";
p = lst1.begin ();
while(p != lst1.end()) { cout << *p;
                         p++; }
return 0;}
```

☞ The lists contains the letters ***ACEGI*** and ***BDFHJ***, are merged to produce the sequence ***ABCDEFGHIJ***.

14.12 MAPS (example of an *associative container*)

The ***map class*** supports an ***associative container*** in which ***unique keys*** are mapped with values. Generally a map is a ***list of key/value pairs***. In essence, a ***key*** is simply a ***name*** that you give to a ***value***. Once a ***value*** has been ***stored***, you can ***retrieve*** it by using its ***key***.

- A ***map*** can hold only ***unique*** keys. ***Duplicate keys are not allowed***. To create a map that allows ***nonunique*** keys, use ***multimap***.
- The power of a ***map*** is that you can look up a value given its ***key***. For example, you could define a ***map*** that uses a ***person's name*** as its ***key*** and stores that ***person's telephone number*** as its ***value***.

❒ **`template` *specification for* `map`*:***

`template <class Key, class T, class Comp=less<Key>, class Allocator=allocator<T> > class map`

- Here ***`Key`*** is the ***data-type*** of the ***keys***.
- ***`T`*** is the data-***`type`*** of the values being ***stored (mapped)***
- ***`Allocator`*** specifies the allocator, which defaults to the **standard allocator**
- ***`Comp`*** is a function that compares **two keys**. This defaults to the standard ***`less()`*** ***utility function*** object.
- These ***comparison operators*** are defined for ***map***: **`==, <, <=, !=, >, >=`**

❒ ***map*** has the following ***constructors*** :

[1] This form constructs an ***empty map :***

```
explicit map(const Comp &cmpfn=Comp(), const Allocator &a=Allocator() );
```

[2] The second form constructs a ***map*** that contains the **same elements** as ***ob***.

```
map( const map<key, T, Comp, Allocator> &ob);
```

[3] The third form constructs a ***map*** that contains the **elements in the range** specified by the ***iterators*** ***`start`*** and ***end***. (This constructor uses **specific instance**):

***template** <**class InIter**> map(**InIter** start, **InIter** end, **const** Comp &cmpfn=Comp(), **const Allocator** &a= Allocator());*

☞ The function specified by ***cmpfn***, if present, **determines the ordering** of the ***map***.

❒ Any ***object*** (or ***object-type***) used as a ***key*** must define a ***default constructor*** and ***overload*** ***any necessary*** ***comparison operators*** (i.e. `==, <=, <, >` etc) as we did for ***`vectors`*** & ***`list`*** (see ***vector 14.10: Example 4***).

❒ The ***member functions*** defined by ***map*** are shown in following **Table**. In the descriptions, ***`key_type`*** is the type of the ***key*** and ***`value_type`*** represents ***`pair<Key, T>`***.

❒ ***Key/value pairs,*** `pair()` ***template class :*** ***Key/value pairs*** are stored in a ***map*** as **objects of type** ***pair***. Template specification:

```
template <class Ktype, class Vtype> struct pair{
                        typedef Ktype first_type ;          /* type of key */
                        typedef Vtype second_type ;         /* type of value */

                        Ktype first ;                       /* contains the key */
                        Vtype second ;                      /* contains the value */
        pair();                                             /* constructor 1 */
        pair(const Ktype &k, const Vtype &v);               /* constructor 2 */
        template<class A, class B> pair(const<A, B> &ob);   /* constructor 3 */   }
```

☞ the value in ***first*** contains the ***key*** and the value in ***second*** contains the ***value*** associated with that key.

❒ ***make_pair() GnF:*** To construct a ***pair*** use either one of ***pair's constructors*** (above 3 constructors) or use ***make_pair()***, which constructs *a pair object based upon the types of the data used as parameters.* ***make_pair()*** is a ***GnF*** that has following prototype:

```
template< class Ktype, class Vtype>
        pair<Ktype, Vtype> make_pair(const Ktype &k, const Vtype &v);
```

➲ it returns a ***pair object*** consisting of values of the types specified by ***Ktype*** and ***Vtype***.

➲ The advantage of ***make_pair(),*** it allows the ***types of the objects*** being stored to be **determined automatically** by ***compiler***.

Member Function		***Description***
iterator *begin*();	**const_iterator** *begin*() **const**;	**Returns an iterator** to the ***first*** element in the map.
iterator *end*();	**const_iterator** *end*() **const**;	**Returns an iterator** to the ***end*** of the map.
iterator *find*(**const** key_type &k); **const_iterator** *find*(**const** key_type &k) **const**;		**Returns an iterator** to the ***specified key***. If the key is not found, an iterator to the ***end*** of the map is returned.
iterator *lower_bound*(**const** key_type &k); **const_iterator** *lower_bound*(**const** key_type &k **const**;		**Returns an iterator** to the ***first*** element in the map with key equal to or greater than ***k***.
iterator *upper_bound*(**const** key_type &k); **const_iterator** *upper_bound*(**const** key_type &k) **const**;		**Returns an iterator** to the ***first*** element in the map with the key greater than ***k***.
reverse_iterator *rbegin*();	**const_reverse_iterator** *rbegin*() **const**;	Returns a **reverse iterator** to the ***end*** of the map.
reverse_iterator *rend*();	**const_reverse_iterator** *rend*() **const**;	Returns a reverse iterator to the start of the map.
void *erase*(**iterator** i);		**Removes** the element ***pointed*** to by ***i***.
void *erase*(**iterator** start, **iterator** end);		**Removes** the elements in the range ***start*** to ***end***.
size_type *erase*(**const** key_type &k);		**Removes** elements that have keys with the value ***k***.
value_compare *value_comp*() **const**;		**Returns** the ***function object*** that ***compares*** values.
key_compare *key_comp*() **const**;		**Returns** the ***function object*** that ***compares*** keys.
size_type *max_size*() **const**;		Returns the ***max number*** of elements that the ***map*** can hold.
size_type *size*() **const**;		Returns the ***current-number*** of elements in the map.
void *clear*();		Removes all elements from the map.
bool *empty*() **const**;		Returns ***true*** if invoking map is ***empty*** and ***false otherwise***.
allocator_type *get_allocator*() **const**;		Returns the map's allocator.
size_type *count*(**const** key_type &k) **const**;		Returns the **number of times** ***k*** occurs in the ***map*** (1 or 0).
pair<iterator, **iterator**> *equal_range*(**const** key_type &k); **pair<const_iterator**, **const_iterator**> *equal_range*(**const** key_type &k) **const**;		Returns a ***pair of iterators*** that point to the ***first*** and ***last*** elements in the map that contain the *specified key.*
iterator *insert*(**iterator** i, **const** value_type &val);		Inserts ***val*** at or after the element specified by ***i***. ***iterator*** to the element is returned.
pair<iterator, **bool**> *insert*(**const** value_type &val);		Inserts ***val*** into the invoking ***map***. An ***iterator*** to the element is ***returned***. The element is inserted ***only if it does not already exist***. If the element was inserted, ***pair<iterator, true>*** is returned. Otherwise, ***pair<iterator, false>*** is returned.
template <class InIter> void *insert*(**InIter** start, **InIter** end);		Inserts a range of elements.
reference *operator*[ ](**const** key_type &i);		**Returns a reference** to the element specified by ***i***. If element *doesn't exist*, it is ***inserted***.
void *swap*(map<Key, T, Comp, **Allocator**> &ob);		**Exchanges** the **elements** stored in the invoking ***map*** with those in ***ob***.

❒ ***Example 1:*** Following program stores ten ***key/value*** pairs. The key is a character and the value is an integer. Stored ***key/value*** pairs :

```
(A, 0)      (B, 1)      (C, 2)      and so on.
```

❖ The user is prompted for a key (i.e., a letter from ***A*** through ***J***), and the value associated with that key is displayed.

```
#include<iostream>
#include<map>
using namespace std;
int main(){     map <char , int > m;                        /* map declaration */
                int i;
                        /* use pair template-class for key/value pair */
                for(i=0; i<10; i++){ m.insert(pair<char, int>('A'+i, i));}

                char ch;                                    /* accessing part */
                cout << " Enter key : ";
                cin >> ch;

                map<char, int>:: iterator p;                /* iterator declaration */
                p = m.find(ch);                             /* find value given key */

                if(p != m.end()) cout << p-> second ;       /* second is form pairs()'s template specification */
                else cout << "Key not in map. \n";
        return 0; }
```

- ***pair*** template class used to construct the ***key/value pairs***.
- Once the ***map*** has been initialized with ***keys*** and ***values***, you can search for ***value*** given its ***key*** by using the ***find()*** function.
  - ***find()*** returns an **iterator** to the matching element or to the **end of the map** if the key is not found.
  - When a match is found, the ***value*** associated with the ***key*** is contained in the ***second*** member of ***pair***.
- Here, ***key/value*** pairs were constructed explicitly, using ***pair<char, int>***. However, the easier way is to use ***make_pair()*** which constructs a pair object ***based upon the types of the data used as parameters*** (sometimes **type-cast** is needed to prevent ***auto-type-determination feature*** of this function). For example, following code will also insert ***key/value pairs*** into ***m***:

```
m.insert( make_pair(( char )('A'+i), i) );
```

  - Here the ***cast to char*** is needed to ***override*** the ***automatic conversion*** to ***int*** when ***i*** is added to '***A***'. Otherwise, the **type determination is automatic**.

14.13 ALGORITHMS (names of the algorithms with purpose)

Although each container provides support for its own basic operations, the **std algorithms** provide more extended or complex actions.

- Algorithms allow you to work with ***two different types of containers at the same time***. All of the algorithms are ***template functions***. This means that they ***can be applied to any type of container***.
  - To have access to the ***STL*** algorithms, you must include ***<algorithm>*** in your program.
  - The ***STL*** defines a large number of algorithms, which are summarized **alphabetically** in following Table.

Algorithm	***Purpose***
adjacent_find	Searches for adjacent matching elements within a sequence and returns an iterator to the first match.
binary_search	Performs a binary search on an ordered sequence.
copy	Copies a sequence
copy_backward	Same as copy() except that it moves the elements from the end of the sequence first.
count	Returns the number of elements in the sequence.
count_if	Returns the number of elements in the sequence that satisfy some predicate
equal	Determines whether two ranges are the same.
equal_range	Returns a range in which an element can be inserted into a sequence without disrupting the ordering of the sequence.
fill ***fill_n***	Fills a range with the specified value
find	Searches a range for a value and returns an iterator to the first occurrence of the element.
find_end	Searches a range for a subsequence. This function returns an iterator to the end of the subsequence within the range.
find_first_of	Finds the first element within a sequence that matches an element within a range.
find_if	Searches a range for an element for which a user-defined unary predicate returns true.
for_each	Applies a function to a range of elements.
generate ***generate_n***	Assigns to elements in a range the values returned by a generator function.
includes	Determines whether one sequence includes all of theelements all of the elements in another sequence.
inplace_merge	Merges a range with another range. Both ranges must be sorted in increasing order. The resulting sequence is sorted.
iter_swap	Exchanges the values pointed to by its two iterator arguments.
lexicographical_compare	Alphabetically compares one sequence with another.
lower_bound	Finds the first point in the sequence that is not less than a specific value.
make_heap	Constructs a heap from a sequence.
max	Returns the maximum of two values.
max_element	Returns an iterator to the maximum element within a range
merge	Merges two ordered sequences, placing the result into a third sequence
min_element	Returns an iterator to the minimum element within a range.
mismatch	Finds the first mismatch between the elements in two sequences. Iterators to the two elements are returned.
next_permutation	Constructs the next permutation of a sequence.
nth_element	Arranges a sequence such that all elements less than a specified element E come before that element and all elements greater than E come after it,
partial_sort	Sorts a range.
partial_sort_copy	Sorts a range and then copies as many elements as will fit into a result sequence.
partition	Arranges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns false.
pop_heap	Exchanges the first and last -1 elements and then rebuilds the heap.
prev_permutation	Constructs the previous permutation of a sequence.
push_heap	Pushes an element onto the end of a heap.
remove ***remove_if***	***remove_copy*** ***remove_copy_if*** Removes elements from a specified range.
replace ***replace_if***	***replace_copy*** ***replace_copy_if*** Replaces elements within a specified range.
reverse ***reverse_copy***	Reverses the order of a range.
rotate ***rotate_copy***	Left-rotates the elements in a range.
search	Searches for a subsequence within a sequence
search_n	Searches for a sequence of a specified number of similar elements.
set_difference	Produces a sequence that contains the difference between ordered sets.
set_intersection	Produces a sequence that contains the intersection of two ordered sets.
set_symmetric_difference	Produces a sequence that contains the symmetric difference between two ordered sets.
set_union	Produces a sequence that contains the union of two ordered sets.
sort	Sorts a range.

sort_heap	Sorts a heap within a specified range.
stable_partition	Arranges a sequence such that all elements for which a predicate returns true come before those for which the predicate returns false. The partitioning is stable; the relative ordering of the sequence is preserved.
stable_sort	Sorts a range. The sort is stable; equal elements are not rearranged.
swap	Exchange two values.
swap_ranges	Exchanges elements in a range
transform	Applies a function to a range of elements and stores the outcome in a new sequence.
unique ***unique_copy***	Eliminates duplicate elements from a range.
upper_bound	Finds the last point in a sequence that is not greater than some value.

Generic Name	*Represents*	*Generic Name*	*Represents*	*Generic Name*	*Represents*
BiIter	Bidirectional iterator	***RandIter***	Random access iterator	***Generator***	A function that generates objects
ForIter	Forward iterator	***T***	Some type of data	***BinPred***	Binary predicate
InIter	Input iterator	***Size***	Some type of integer	***UnPred***	Unary predicate
OutIter	Output iterator	***Func***	Some type of function	***Comp***	Comparison function

❒ ***count() and count_if():*** Two of the simplest algorithms are ***count()*** and ***count_if()***. Their general forms are :

```
template <class InIter, class T> size_t count(InIter start, InIter end, const T &val );
template <class InIter, class UnPred> ptrdiff_t count_if(InIter start, InIter end, UnPred pfn);
```

- The ***count()*** algorithm returns the number of elements in the sequence beginning at ***start*** and ending at ***end*** that match ***val***.
- The ***count_if()*** algorithm returns the number of elements in the sequence beginning at ***start*** and ending at ***end*** for which the ***unary predicate pfn*** returns ***true***.

☞ The following program demonstrates ***count()*** and ***count_if()***.

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

/* This is a unary predicate that determines if a value is even. 1 for even, 0 for odd.*/
bool even(int x){ return !(x %2); }

int main(){ vector <int> v;
            int i;
              /* putting 1's and 2's on the vector */
            for(i=0; i <20; i++){ if(i%2) v.push_back(1);
                                  else v.push_back(2); }
cout << " Sequence : ";                    /* display the vector */
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl ;

int n;                                     /* count and count_if */
n = count(v.begin(), v.end(), 1);
cout << n << " elements are 1\n";

n = count_if(v.begin() , v.end(), even);
cout << n << " elements are even .\n";
return 0; }
```

OUTPUT:
```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
10 elements are 1
10 elements are even.
```

☞ At the start we create a ***20-element vector*** that contains alternating ***1s*** and ***2s***. Notice how the unary predicate ***even()*** is coded.

☞ ***count()*** is used to count the number of ***1's***. Then, ***count_if()*** counts the number of elements that are ***even***.

☞ ***Unary predicate rules:*** All unary predicates receive as a parameter an object that is of the same type as that stored in the container upon which the predicate is operating. The predicate must then return a true or false result based upon this object.

❒ ***remove_copy():*** To generate a **new sequence** that consists of only certain items from an **original sequence** use ***remove_copy()***

template <class InIter, class OutIter, class T> OutIter remove_copy(InIter start, InIter end, OutIter result, const T &val);

- The ***remove_copy()*** algorithm copies elements from the specified range that are equal to ***val*** and puts the result into the sequence pointed to by result.
- It ***returns an iterator*** to the ***end*** of the result. The output container must be large enough to hold the result.

☞ following demonstrates ***remove_copy()***. It creates a sequence of ***1's*** and ***2's***. Then removes all of the ***1's*** from the sequence.

```
int main(){vector <int> v, v2 (20);
           int i;
           for(i=0; i<20; i++){if(i%2)v.push_back(1);
                               else v.push_back(2);}   /* putting 1's and 2's */
           cout << "Sequence : ";
           for(i=0; i<v.size(); i++) cout << v[i] << " ";   /* original */
           cout << << endl;
remove_copy(v.begin(), v.end(), v2.begin(), 1);   /* Remove 1s */
cout << " Result : ";
for(i=0; i<v2.size(); i++) cout << v2[i] << " " << endl << endl ;
return 0; }
```

OUTPUT:
```
Sequence: 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1
Result: 2 2 2 2 2 2 2 2 2 2 0 0 0 0 0 0 0 0 0 0
```

❒ ***reverse():*** An often useful algorithm is ***reverse()***, which **reverses a sequence**. Its general form is:

```
template <class BiIter> void reverse(BiIter start, BiIter end);
```

- The ***reverse()*** algorithm reverses the order of the range specified by ***start*** and ***end***. Example:

```
int main(){ vector <int> v;
            int i;
            for(i=0; i<10; i++) v.push_back(i);
            cout << " Initial : ";
            for(i=0; i<v.size(); i++) cout << v[i] << " " << endl ;
reverse(v.begin(), v.end());  /* reversing the sequence */
cout << " Reversed : ";
for (i=0; i<v.size(); i++) cout << v[i] << " ";
return 0; }
```

❒ ***transform():*** ***transform() modifies each element in a range*** according to a function that you provide. It has two general forms:

☞ The ***transform()*** algorithm applies a function to a range of elements and stores the outcome in result.

☞ Both versions return an iterator to the end of the resulting sequence.

```
template <class InIter, class OutIter, class Func>
        OutIter transform(InIter start, InIter end, OutIter result, Func unaryfunc);
```

- In this form, the range is specified by ***start*** and ***end***. The function to be applied is specified by ***unaryfunc***.
- This function receives the value of an element in its parameter, and it must return the element's transformation.

template <**class InIter1**, **class InIter2**, **class OutIter**, **class** Func>
OutIter *transform*(**InIter1** start1, **InIter1** end1, **InIter2** start2, **OutIter** result, Func *binaryfunc*);

- In this form, the ***transformation*** is applied using a **binary operator function** that receives the value of an element from the sequence to be transformed in its ***first parameter*** and an element from a ***second sequence*** as its ***second parameter***.

```
                    /* A simple transformation function.*/
int xform(int i){ return i*i; }                    /* square original value */
int main(){list <int> x1;
        int i;
        for (i=0; i<10; i++) x1.push_back(i);      /* putting the values */
        cout << "Original contents of x1:";
        list <int> :: iterator p = x1.begin();     /* iterator declare with initialization */
        while(p != x1.end()) {cout << *p << " "; p++; } /* display original */
        cout << endl ;
        p = transform(x1.begin(), x1.end(), x1.begin(), xform); /* transformation */
        cout << " Transformed x1: ";
        p = x1.begin();        /* iterator initialization */
        while (p != x1.end()) { cout << *p << " "; p++; }
        return 0; }
```

OUTPUT;

```
Original contents of x1: 0 1 2 3 4 5 6 7 8 9
Transformed  x1: 0 1 4 9 16 25 36 49 64 81
```

☞ As you can see, each elements in the ***x1*** list has been ***squared***.

14.14 STRING class

C++ does not support a **built-in string type**. However, C++ provide ***two ways*** to handle strings.

[1] First, you can use the traditional, ***null-terminated character array*** which we usually use. This is sometimes referred to as a ***C string***.

[2] The second method is to use a ***class object*** of ***type string***. This approach is examined here.

❒ ***String class*** is a specialization of a ***more general template class*** called ***basic_string***. two specializations of ***basic_string*** :
- ▶ ***string***, which supports ***8-bit character strings***,
- ▶ ***wstring***, which supports ***wide character strings***.

☞ Since 8-bit characters are most commonly used we'll focus on ***string*** here.

❒ ***why string class is part of the C++ library:*** Since C++ already contains some support for strings as null-terminated character arrays, it might at first seem that the inclusion of the string class is an exception to this rule. There are ***three reasons*** for the inclusion of the ***standard string class***

☛ ***Consistency:*** Using **standard string class** a ***string*** now defines a ***data-type***.

☛ ***Convenience(you can use the standard C++ operators):*** ***Null-terminated strings*** cannot be manipulated by any of the ***standard C++ operators***, nor can they take part in ***normal C++ expressions***. For example:

```
char s1[80], s2[80], s3[80];
s1 = "one";                 /* can 't do */
s2 = "two";                 /* can 't do */
s3 = s1 + s2;               /* error , not allowed */
```

- ▶ In C++ it is not possible to use the ***assignment operator*** to give a character array a new value (except during initialization),
- ▶ Also it is not possible to use the **+** operator to ***concatenate*** two strings. These must be written using library functions as:

```
strcpy(s1, "one");    strcpy(s2, "two");    strcpy(s3, s1);       strcat(s3, s2);
```

- ▶ Since ***null-terminated character arrays*** are not technically ***data types***, the C++ operators cannot be applied to them. This inability of operation has driven the development of a ***standard string class***. By **standard string class** we can apply C++ operators on strings (just like other **data types**).

☛ ***safety(array boundaries will not be overrun):*** ***Safety*** is one other reason for the standard string class. An inexperience or careless programmer can very easily **overrun** the end of an array that holds a ***null-terminated string***. For example, ***strcpy()*** contains no provision for checking the boundary of the target array. If the source array contains more characters than the target array can hold, a program error/system-crash occurs.

★ However, for speedy program simple ***null-terminated strings*** is a good choice.

❒ ***Overload the operators for new class-types:*** Remember, when you define a class in C++, you are defining a new data type that can be fully integrated into the C++ environment. This, of course, means that the operators can be overloaded relative to the new class.

❒ Although not traditionally thought of as part of the ***STL***, string is another container class defined by C++. This means that it supports the ***algorithms*** described in the previous section.

☞ To have access to the ***string class*** you must include **<string>**. The string class supports several ***constructors***. The prototypes for three of its most ***commonly used constructors*** are:
- ➢ This form creates an **empty** string object: ***string();***
- ➢ The second creates a string object from the **null-terminated string pointed** to by ***str***. This form provides a **conversion** from ***null-terminated strings*** to ***string objects***: ***string(const char *str);***
- ➢ The third form creates a string object from another string object: ***string(const string &str);***

☞ A number of operators that apply to strings are defined for string objects, including those listed in the following table:

Operator	**=**	**+**	**+=**	**==**	**!=**	**<**
Meaning	Assignment	Concatenation	Concatenation assignment	Equality	Inequality	Less than
Operator	**<=**	**>**	**>=**	**[ ]**	**<<**	**>>**
Meaning	Less than or equal	Greater than	Greater than or equal	Subscripting	Output	Input

⇨ These operators allow the use of string objects in normal expressions and eliminate the need for calls to functions such as ***strcpy()*** or ***strcat()***.

☞ In general, you can mix string objects with normal, null-terminated strings in expressions. For example, a string object can be assigned a null-terminated string.

☞ The **+** operator can be used to ***concatenate*** a string object with another string object or a string object with a ***C-style string***. i.e.

```
string + string          string + C-string          C-string + string
```

▶ The **+** operator can also be used to ***concatenate*** a character onto the *end of a string*.

☞ The ***string class*** defines the constant ***npos***, which is usually ***-1***, represents the length of the longest possible string.

❒ ***string class member functions:*** Complex string operations are accomplished with string class member functions. Most common string class member functions are discussed below:

☛ **assign()**: To ***assign*** one string to another, use the ***assign()*** function. Two of its forms are:

```
string &assign(const string &strob, size_type start, size_type num);
string &assign(const char *str, size_type num );
```

- In the first form, ***num*** characters from ***strob*** beginning at ***start*** will be ***assigned*** to the invoking object.
- In the second form, the first ***num*** characters of the ***null- terminated string str*** are assigned to the invoking object.
- In each case, a **reference** to the invoking object is *returned*.

☛ **append()**: You can ***append*** part of one string to another using the ***append()***.Two of its forms are:

```
string &append(const string &strob, size_type start, size_type num);
string &append(const char *str, size_type num);
```

- In the first form, ***num*** characters from ***strob***, beginning at ***start***, will be ***appended*** to the invoking object.
- In the second form, the first ***num*** characters of the ***null-terminated string str*** are appended to the invoking object.
- In each case, a **reference** to the invoking object is *returned*.

☛ **insert()** and **replace()**: You can ***insert*** or ***replace*** characters within a string using ***insert()*** and ***replace()***.

```
string &insert(size_type start, const string &strob);
string &insert(size_type start, const string &strob, size_type insStart, size_type num);
```

- The first form of ***insert()*** inserts ***strob*** into the invoking string at the index specified by ***start***.
- The second form of ***insert()*** inserts ***num***, characters from ***strob*** beginning at ***insStart*** into the invoking string at the index specified by ***start***.

```
string &replace(size_type start, size_type num, const string &strob );
string &replace(size_type start, size_type orgNum , const string &strob , size_type replaceStart, size_type replaceNum);
```

- Beginning at ***start***, the first form of ***replace()*** replaces ***num*** characters from the invoking string with ***strob***.
- The second form replaces ***orgNum*** characters, beginning at ***start*** in the invoking string, with ***replaceNum*** characters from the string specified by ***strob*** beginning at ***replaceStart***.
- In both cases, a **reference** to the invoking object is *returned*.

☛ ***erase()***: You can ***remove characters*** from a string using ***erase()***. One of its forms is:

```
string &erase(size_type start=0, size_type num=npos);
```

- It removes ***num*** characters from the invoking string beginning at ***start***. A **reference** to the invoking string is *returned*.

☛ ***find()*** and ***rfind()***: ***find()*** and ***rfind()***,search a string. (There are also other functions for searching.)

```
size_type find(const string &strob , size_type start =0) const;
```

- Beginning at ***start***, ***find()*** searches the invoking string for the first occurrence of the string contained in ***strob***.
- If the search string is found, ***find()*** returns the index at which the match occurs within the invoking string. If no match is found, ***npos*** is returned.

```
size_type rfind(const string &strob , size_type start = npos ) const;
```

- ***rfind()*** is the opposite of ***find()***. Beginning at ***start***, it searches the invoking string in the ***reverse direction*** for the first occurrence of the string contained in ***strob***. (i.e., it finds the ***last occurrence*** of ***strob*** within the invoking string.)
- If the search string is found, ***rfind()*** returns the index at which the match occurs within the invoking string. If no match is found, ***npos*** is returned.

☛ ***compare()***: To ***compare*** the entire contents of one string object to those of another, you will normally use the overloaded relational operators described earlier. However, if you want to compare a portion of one string to another, use ***compare()***

```
int compare(size_type start, size_type num, const string &strob) const;
```

- Here ***num*** characters in ***strob***, beginning at ***start***, will be compared against the invoking string.
- If the invoking string is less than ***strob***, ***compare()*** will return ***0***.

☛ ***c_str()*** *for null-terminated character array version of the string:* Although ***string objects*** are, there will be times when you will need to obtain a ***null-terminated character array version of the string***. For example, you might use a ***string object*** to *construct a file name*. However, when *opening a file*, you will need to specify a ***pointer*** to a ***standard null-terminated string***. To solve this problem, the member function ***c_str()*** is provided. Its prototype is:

```
const char *c_str() const ;
```

- This function ***returns*** a ***pointer*** to ***a null-terminated version of the string*** contained in the invoking string object.
- The ***null-terminated string*** must not be altered. It is also not guaranteed to be valid after any other operations have taken place on the string object.

☞ ***begin()***, ***end()*** and ***size()***: Since string is a container, it supports the ***begin()*** and ***end()*** functions that ***return*** an ***iterator*** to the ***start*** and ***end*** of a string, respectively.
- ❖ Also included is the ***size()*** function, which returns the ***number of characters*** currently in a string.

❒ ***Example 1:*** C++ string class makes string handling easy. For example, with string objects you can use the ***assignment operator*** to assign a quoted string to a string, the ***+ operator*** to concatenate strings, and the ***comparison operators*** to compare strings.

```
#include <iostream>
#include <string>
using namespace std;
int main() { string str1(" Demonstrating Strings ");
        string str2 (" String Two ");
        string str3 ;
        str3 = str1;                          /* assign a string */
        cout << str1 << "\n" << str3 << "\n";

        str3 = str1 + str2 ;                  /* concatenate two strings */
        cout << str3 << "\n";

        if(str3 > str1) cout << " str3 > str1 \n";        /* compare strings */
        if(str3 == str1 + str2) cout << " str3 == str1 + str2 \n";

                /* A string object can also be assigned a normal string. */
        str1 = " This is a normal string .\n";
        cout << str1 ;
```

```
/* create a string object using another string object */
        string str4 ( str1 );
        cout << str4 ;

        cout << " Enter a string : ";
        cin >> str4 ; /* input a string */
        cout << str4 ;
return 0; }
```

OUTPUT:

```
Demonstrating Strings
Demonstrating Strings
Demonstrating StringsString Two
str3 >str1
str3 == str1+str2
This is a normal string.
This is a normal string.
Enter a string: Hello
Hello
```

☞ NOTICE: the ***sizes*** of the strings are not specified. A string object is ***automatically sized*** to hold the string that it is given.

❒ ***Example 2:*** The following program demonstrates the ***insert()***, ***erase()***, and ***replace()*** functions.

```
int main() { string str1("This is a test");
        string str2 (" ABCDEFG ");
        cout << "Initial strings :\n";
        cout << "str1 : " << str1 << endl ;
        cout << "str2 : " << str2 <<"\n\n";

       /* demonstrate insert() */
cout << "Insert str2 into str1 :\n";
str1.insert(5, str2 );
cout << str1 << "\n\n";
```

```
       /* demonstrate erase() */
cout << "Remove 7 characters from str1 :\n";
str1.erase(5, 7);
cout << str1 << "\n\n";

       /* demonstrate replace() */
cout << "Replace 2 characters in str1 with
str2 :\n";
str1.replace(5, 2, str2 );
cout << str1 << endl ;
return 0; }
```

OUTPUT:

```
Initial strings:
str1: This is a test
str2: ABCDEFG
Insert str2 into str1:
This ABCDEFGis a test
Remove 7 characters from str1:
This is a test
Replace 2 characters in str1 with str2:
This ABCDEFG a test
```

❒ ***Example 3:*** Since string defines a ***data type***, it is possible to ***create containers that hold objects of type string***. For example,

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() { map<string, string> m;
        int i;

        m.insert( pair<string, string>("yes", "no") );
        m.insert( pair<string, string>("up", " down") );
        m.insert( pair<string, string>("left", "right") );
        m.insert( pair<string, string>("good", "bad") );
```

```
string s;
cout << " Enter word : ";
cin >> s;

map<string, string> :: iterator p;
p = m.find(s);
    if(p != m.end()) cout << "Opposite :" << p -> second ;
    else cout << " Word not in map \n";
return 0;}
```

Finishing Tips (C)

☞ Now that you have finished this book, go back and skim through each chapter, thinking about how each aspect of C relates to the rest of it. As you will see, ***C is a highly integrated language***, in which one feature complements another. ***The connection between pointers and arrays***, for example, is pure elegance.

☞ ***C is a language best learned by doing!*** Continue to ***write programs in C*** and to ***study other programmer's programs***. You will be surprised at how quickly C will become second nature!

☞ Finally, you now have the necessary foundation in C to allow you to ***move on to C++, C's object-oriented extension***. If C++ programming is in your future, proceed to Teach Yourself C++, It picks up where this book leaves off.

Finishing Tips (C++)

☞ You have come a long way since Chapter 1. Take some time to skim through the book again. As you do so, think about ways you can improve the examples (especially those in chapters 9, 10, 11) so that they take advantage of all the features of C++ .

☞ Programming is learned best by doing. Write many C++ programs. Try to exercise those features of C++ that are unique to it.

☞ Continue to explore the ***STL***. In the future, many programming tasks will be ***framed in terms of the STL*** because often a program can be reduced to manipulations of containers by algorithms.

☞ Finally, remember: C++ gives you unprecedented power. It is important that you learn to use this power wisely. Because this power, C++ lets you push the limits of your programming ability. However, if this power is misused, you can also create programs that are hard to understand, nearly impossible to follow, and extremely difficult to maintain. C++ is a powerful too. But, like any other tool, it is only as good as the person using it.