Function Overloading

- In C++, two or more functions can share the same name as long as their parameter declarations are different.
- In this situation, the functions that share the same name are said to be *overloaded*, and the process is referred to as *function overloading*

• To see why function overloading is important, first consider three functions defined by the C subset: **abs(), labs()**, and **fabs().** The **abs()** function returns the absolute value of an integer, **labs()** returns the absolute value of a **long**, and **fabs()** returns the absolute value of a **double**.

• Although these functions perform almost identical actions, in C three slightly different names must be used to represent these essentially similar

tasks.

•Example: Function Overloading

Function Overloading

- Function overloading is the process of using the same name for two or more functions.
- The secret to overloading is that each redefinition of the function must use either-

- different types of parameters
- different number of parameters.

Function Overloading and Ambiguity

- Ambiguous statements are errors, and programs containing ambiguity will not compile.
- By far the main cause of ambiguity involves C++'s automatic type conversions.

Function Overloading and Ambiguity

- void f(int x);
- void f(int &x); // error

- two functions cannot be overloaded when the only difference is that one takes a reference parameter and the other takes a normal, call-by-value parameter.

- // This program contains an error.
- #include <iostream>
- using namespace std;
- void f(int x);
- void f(int &x); // error
- int main()
- {
- int a=10;
- f(a); // error, which f()?
- return 0;
- }
- void f(int x)
- {
- cout << "In f(int)\n";
- }
- void f(int &x)
- {
- cout << "In f(int &)\n";
- }

- char myfunc(unsigned char ch);
- char myfunc(char ch);
- In C++, **unsigned char** and **char** are *not* inherently ambiguous.

- #include <iostream>
- using namespace std;
- char myfunc(unsigned char ch);
- char myfunc(char ch);
- int main()
- {
- cout << myfunc('c'); // this calls myfunc(char)
- cout << myfunc(88) << " "; // ambiguous
- return 0;
- }
- char myfunc(unsigned char ch)
- {
- return ch-1;
- }
- char myfunc(char ch)
- {
- return ch+1;
- }

- #include <iostream>
- using namespace std;
- float myfunc(float i);
- double myfunc(double i);
- int main()
- {
- cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
- cout << myfunc(10); // ambiguous
- return 0;
- }
- float myfunc(float i)
- {
- return i;
- }
- double myfunc(double i)
- {
- return -i;
- }

Others

- Typedef int integer;
- Enum days{mon,tue,wed}
- Void f(int);
- Void f(mon);

Overloading Constructor Functions

- Many times you will create a class for which there are two or more possible ways to construct an object.
- In these cases, you will want to provide an overloaded constructor function for each way.
- The user is free to choose the best way to construct an object given the specific circumstance.

Copy Constructors

- By default, when one object is used to initialize another.
- C++ performs a bitwise copy.
- For Example: MyClass B= A;
- If a bitwise copy is performed, then *B* will be an exact copy of *A*. This means that *B* will be using the same piece of allocated memory that *A* is using, instead of allocating its own.
- If *MyClass* includes a destructor that frees the memory, then the same piece of memory will be freed twice when *A* and *B* are destroyed!

Copy Constructor…….

- The same type of problem can occur in two additional ways:
- first, when a copy of an object is made when it is passed as an argument to a function;
- second, when a temporary object is created as a return value from a function.
- To solve the type of problem just described, C++ allows you to create a *copy constructor*, which the compiler uses when one object initializes another.

- The most common general form of a copy constructor is
  - classname (const *classname &o*) {
  - *// body of constructor*
  - }
- Here, *o* is a reference to the object on the right side of the initialization.
- It is permissible for a copy constructor to have additional parameters as long as they have default

  arguments defined for them.
- However, in all cases the first parameter must be a

  reference to the object doing the initializing.

- It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another.
- The first is assignment.
- The second is initialization, which can occur any of three ways:
    - myclass x = y; // y explicitly initializing x
    - func(y); // y passed as a parameter
    - y = func(); // y receiving a temporary, return object

Example: copy_cons1

- The copy constructor is called, memory for the new array is allocated and stored in **x.p,** and the contents of **num** are copied to **x**'s array.
- In this way, **x** and **num** have arrays that contain the same values, but each array is separate and distinct.
- If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num** sharing the same memory for their arrays.
- array a(10);
- // ...
- array b(10);
- b = a; // does not call copy constructor

Finding the Address of an Overloaded Function

- You can obtain the address of a function.
- Assign the address of the function to a pointer and then call that function through that pointer
- When you assign the address of an overloaded function to a function pointer, it is the declaration of the pointer that determines which function's address is obtained.
- Further, the declaration of the function pointer must exactly match one and only one of the overloaded function's declarations.
- Example:

Default Function Arguments

- C++ allows a function to assign a parameter a default value when no argument corresponding to that parameter is specified in a call to that function.
- The default value is specified in a manner syntactically similar to a variable initialization

Example:

```
void myfunc(double d = 0.0)
{
// ...
}
```

- Now, **myfunc()** can be called one of two ways, as the following examples show:
  - myfunc(198.234); // pass an explicit value
  - myfunc(); // let function use default
- The first call passes the value 198.234 to **d**.
- The second call automatically gives **d** the default value zero.

- There are two advantages to including default arguments, when appropriate, in a constructor function.
- First, they prevent you from having to provide an overloaded constructor that takes no parameters.
- Second, defaulting common initial values is more convenient than specifying them each time an object is declared.
- In some situations, default arguments can be used as a shorthand form of function overloading.

Operator Overloading.

- In C++, you can overload most operators so that they perform special operations relative to classes that you create.
- For example, a class that maintains a stack might overload **+** to perform a push operation and – – to perform a pop.

Overloadable operators.

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

- You overload operators by creating operator functions.
- An *operator function* defines the operations that the overloaded operator will perform relative to the class upon which it will work.
- An operator function is created using the keyword **operator**.
- Operator functions can be either members or nonmembers of a class.
- Nonmember operator functions are almost always friend functions of the class.

Creating a Member Operator Function

- A member operator function takes this general form:
    - *ret-type class-name::*operator#*(arg-list)*
    - {
    - // operations
    - }

- Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type.
- The **#** is a placeholder. When you create an operator function, substitute the operator for the **#**.
- For example, if you are overloading the / operator, use **operator/.**
- When you are overloading a unary operator, *arg-list* will be empty.
- When you are overloading binary operators, *arg-list* will contain one parameter.

of the Increment and Decrement Operators

- *In older versions of C++, it was not possible to specify separate prefixand postfixversions of an overloaded* **++** *or* **– –***.*
- *The prefix form was used for both.*
- The posfix is declared like this:
  - loc operator++(int x);
- If the **++** precedes its operand, the **operator++()** function is called.
- If the **++** follows its operand, the **operator++(int x)** is called and **x** has the value zero.

Decrement

```
// Prefix decrement
type operator– –( ) {
// body of prefix operator
}
// Postfix decrement
type operator– –(int x) {
// body of postfix operator
}
```

Overloading the Shorthand Operators

- You can overload any of C++'s "shorthand" operators, such as **+=, –=,** and the like.
- For example, this function overloads **+=** relative to **loc**:
  - loc loc::operator+=(loc op2)
  - {
  - longitude = op2.longitude + longitude;
  - latitude = op2.latitude + latitude;
  - return *this;
  - }

Operator Overloading Restrictions

- You cannot alter the precedence of an operator.
- You cannot change the number of operands that an operator takes.
- Except for the function call operator functions cannot have default arguments.
- Finally, these operators cannot be overloaded:

  .   .*   ::   ?:   sizeof

Friend operator funcions

- There are some restrictions that apply to friend operator functions.
- First, you may not overload the **=, ( ), [ ],** or **–>** operators by using a friend function.
- Second, when overloading the increment or decrement operators, you will need to use a reference parameter when using a friend function.

Using a Friend to Overload ++ or – –

- If you want to use a friend function to overload the increment or decrement operators, you must pass the operand as a reference parameter.
- This is because friend functions do not have **this** pointers.

Friend Operator Functions Add Flexibility

- Ob + 100 // valid
- 100 + Ob // invalid
- To allow both *object+integer* and *integer+object*, simply overload the function twice—one version for each situation.
- Thus, when you overload an operator by using two **friend** functions, the object may appear on either the left or right side of the operator.

Overloading new and delete

- The skeletons for the functions that overload **new** and **delete** are shown here:

```
// Allocate an object.
void *operator new(size_t size)
{
/* Perform allocation. Throw bad_alloc on failure.
Constructor called automatically. */
return pointer_to_memory;
}
// Delete an object.
void operator delete(void *p)
{
/* Free memory pointed to by p.
Destructor called automatically. */
}
```

New Delete

- The type **size_t** is a defined type capable of containing the largest single piece of memory that can be allocated. (**size_t** is essentially an unsigned integer.)
- The parameter **size** will contain the number of bytes needed to hold the object being allocated.
- The overloaded **new** function must return a pointer to the memory that it allocates, or throw a **bad_alloc** exception if an allocation error occurs.

Overloading Some Special Operators

- Overloading [ ]
- *type class-name*::operator[](int *i*)
- {
- // . . .
- }
- Technically, the parameter does not have to be of type **int**, but an **operator[ ]()** function is typically used to provide array subscripting, and as such, an integer value is generally used.
- Given an object called **O**, the expression
- O[3] translates into this call to the **operator[ ]()** function:
- O.operator[](3)

Overloading ( )

- Given the overloaded operator function declaration
- double operator()(int a, float f, char *s);
- and an object **O** of its class, then the statement
- O(10, 23.34, "hi");
- translates into this call to the **operator()** function.
- O.operator()(10, 23.34, "hi");
- In general, when you overload the **( )** operator, you define the parameters that you want to pass to that function. When you use the ( ) operator in your program, the

Overloading –>

- The **–>** pointer operator, also called the *class member access* operator, is considered a unary operator when overloading. Its general usage is shown here:
- *object->element;*
- Here, *object* is the object that activates the call. The **operator–>()** function must return a pointer to an object of the class that **operator–>()** operates upon.
- The *element* must be some member accessible within the object.

Overloading the Comma Operator

- If you want the overloaded comma to perform in a fashion similar to its normal operation, then your version must discard the values of all operands except the rightmost.
- The rightmost value becomes the result of the comma operation.

Inheritance

Inheritance

- It allows the creation of hierarchical classifications.
- Using inheritance, you can create a general class that defines traits common to a set of related items.
- This class may then be inherited by other, more specific classes, each adding only those things that are unique to the inheriting class.

Inheritance

- A class that is inherited is referred to as a *base class*.
- The class that does the inheriting is called the *derived class*.
- Further, a derived class can be used as a base class for another derived class. In this way, multiple inheritance is achieved.

Base-Class Access Control

- When a class inherits another, the members of the base class become members of the derived class.
- Class inheritance uses this general form:

```
class derived-class-name : access base-class-name {
// body of class
};
```

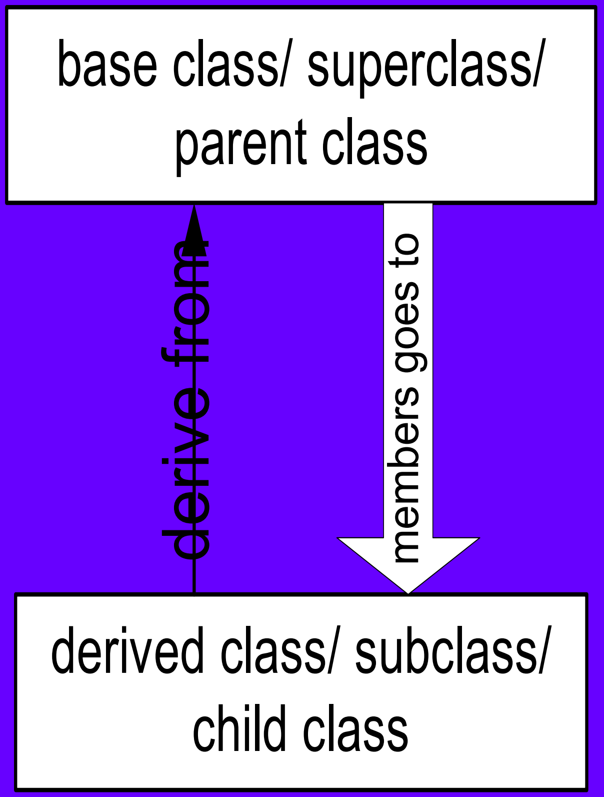
- The access status of the base-class members inside the derived class is determined by *access*.

- The base-class access specifier must be either **public, private,** or **protected**.
- If no access specifier is present, the access specifier is **private** by default if the derived class is a **class**.
- If the derived class is a **struct**, then **public** is the default in the absence of an explicit access specifier.

What to inherit?

- In principle, every member of a base class is inherited by a derived class
  - just with different access permission

Access Control Over the Members


base class/ superclass/ parent class
derive from
members goes to
derived class/ subclass/ child class

- Two levels of access control over class members
  - class definition
  - inheritance type

```
class Point{
    protected: int x, y;
    public: void set(int a, int b);
};
```

```
class Circle : public Point{
    … …
};
```

Even more …

- A derived class can override methods defined in its parent class. With overriding,
  - the method in the subclass has the identical signature to the method in the base class.
  - a subclass implements its own version of a base class method.

```
class A {
  protected:
    int x, y;
  public:
    void print ()
            {cout<<“From A”<<endl;}
};
```

```
class B : public A {
  public:
    void print ()
        {cout<<“From B”<<endl;}
};
```

Private, Public, and Protected Base Classes

- A base class may be specified to be private, public, or protected. Unless so specified, the base class is assumed to be private:

```
class A {
   private:
   int x;
   void Fx (void);
   public:
   int y;
   void Fy (void);
   protected:
   int z;
   void Fz (void);
};
```

```
class B : A {};     // A is a private base class of B
class C : private A {};      // A is a private base
                                  class of C
```

- The behavior of these is as follows :
- All the members of a private base class become *private* members of the derived class. So x, Fx, y, Fy, z, and Fz all become private members of B and C.

```
class D : public A {};          // A is a public base class of D
```

- The behavior of these is as follows :
- The members of a public base class keep their access characteristics in the derived class. So, x and Fx becomes private members of D, y and Fy become public members of D, and z and Fz become protected members of D.

```
class E : protected A {};        // A is a protected
                                    base class of E
```

- The behavior of these is as follows :
- The private members of a protected base class become *private* members of the derived class. Whereas, the public and protected members of a protected base class become *protected* members of the derived class. So, x and Fx become private members of E, and y, Fy, z, and Fz become protected members of E.

Base class access inheritance rules.

Base Class	**Private Derived**	**Public Derived**	**Protected Derived**
Private Member	*private*	*private*	*private*
Public Member	*private*	*public*	*protected*
Protected Member	*private*	*protected*	*protected*

Private access specifier

- When the base class is inherited by using the **private** access specifier, all public and protected members of the base class become private members of the derived class.Example:
- *When a base class' access specifier is **private,** public and protected members of the base become private members of the derived class.*
- *This means that they are still accessible by members of the derived class but cannot be accessed by parts of your program that are not members of either the base or derived class.*

- It is also possible to individually exempt a base class member from the access changes specified by a derived class, so that it retains its original access characteristics. To do this, the exempted member is fully named in the derived class under its original access characteristic. For example:

```
class C : private A {
   //...
public:                A::Fy;  // makes Fy a public member of C
protected:A::z;               // makes z a protected member of C
};
```

Inheritance and protected Members

- When a member of a class is declared as **protected**, that member is not accessible by other, nonmember elements of the program.
- With one important exception, access to a protected member is the same as access to a private member—it can be accessed only by other members of its class. The sole exception to this is when a protected member is inherited.

Protected in Public Inheritance

- When a derived class is used as a base class for another derived class, any protected member of the initial base class that is inherited (as public) by the first derived class may also be inherited as protected again by a second derived class.

Protected Base-Class Inheritance

- It is possible to inherit a base class as **protected.**
- When this is done, all public and protected members of the base class become protected members of the derived class.
- For example,

Inheriting Multiple Base Classes

- It is possible for a derived class to inherit two or more base classes.
- To inherit more than one base class, use a comma-separated list.
- Further, be sure to use an access-specifier for each base inherited.

Multiple Inheritance

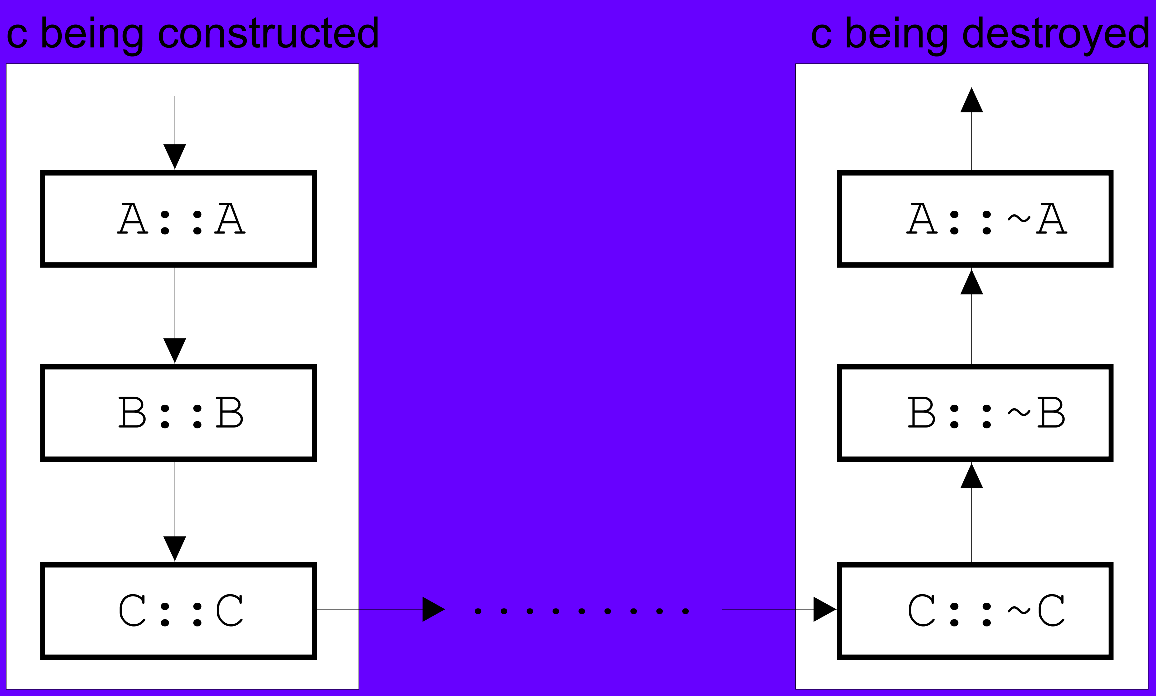
- Multiple inheritance
  - Derived class has several base classes
  - Powerful, but can cause ambiguity problems
    - If both base classes have functions of the same name
    - Solution: specify exact function using `::`
      - `myObject.BaseClass1::function()`
  - Format
    - Use comma-separated list

```
class Derived : public Base1, public Base2{
    contents
}
```

Constructors and Destructors

- When an object of a derived class is created, the base class constructor is applied to it first, followed by the derived class constructor.
- When the object is destroyed, the destructor of the derived class is applied first, followed by the base class destructor.
- Constructors are applied in order of derivation and destructors are applied in the reverse order.

- class A { /* ... */ }
- class B : public A { /* ... */ }
- class C : public B { /* ... */ }



Constructors: Multi level Inheritance

- In cases of multiple inheritance (that is, where a derived class becomes the base class for another derived class), the general rule applies:
- Constructors are called in order of derivation, destructors in reverse order. For example,

Constructors: Multiple Inheritance

- Constructors are called in order of derivation, left to right, as specified in **derived**'s inheritance list.
- Destructors are called in reverse order, right to left.
- class derived: public base2, public base1

  then the output of this program would have looked like this:

```
Constructing base2
Constructing base1
Constructing derived
Destructing derived
Destructing base1
Destructing base2
```

Passing Parameters to Base-Class Constructors

```
derived-constructor(arg-list) : base1(arg-list),
base2(arg-list),
// ...
baseN(arg-list)
{
// body of derived constructor
}
```

Virtual Base Classes

- An element of ambiguity can be introduced into a C++ program when multiple base classes are inherited.
- Example: Ambiguity
- There are two ways to remedy the preceding program.
- The first is to apply the scope resolution operator to **i** and manually select one **i.**

Virtual Base Classes

- When two or more objects are derived from a common base class, you can prevent multiple copies of the base class from being present in an object derived from those objects by declaring the base class as **virtual** when it is inherited.
- You accomplish this by preceding the base class' name with the keyword **virtual** when it is inherited.
- Inherited **base** as **virtual**, any multiple inheritance involving them will cause only one copy of **base** to be present.

Thus,

- The only difference between a normal base class and a **virtual** one is what occurs when an object inherits the base more than once.
- If **virtual** base classes are used, then only one base class is present in the object.
- Otherwise, multiple copies will be found.

Run Time Polymorphism (Virtual Functions)

- Polymorphism is supported by C++ both at compile time and at run time.
- Compile-time polymorphism is achieved by overloading functions and operators.
- Run-time polymorphism is accomplished by using inheritance and virtual functions.

Virtual Functions

- A *virtual function* is a member function that is declared within a base class and redefined by a derived class.
- To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**.
- When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs.

Virtual Functions

- In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism.
- The virtual function within the base class defines the *form* of the *interface* to that function.
- Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class.

Virtual Functions

- When accessed "normally," virtual functions behave just like any other type of class member function.
- However, what makes virtual functions important and capable of supporting run-time polymorphism is how they behave when accessed via a pointer.

Virtual Functions

- When a base pointer points to a derived object that contains a virtual function, C++ determines which version of that function to call based upon *the type of object pointed to* by the pointer.
- And this determination is made *at run time*.
- Thus, when different objects are pointed to, different versions of the virtual function are executed.

Virtual Function vs Function Overloading

- At first glance, the redefinition of a virtual function by a derived class appears similar to function overloading.
- However, this is not the case, and the term *overloading* is not applied to virtual function redefinition because several differences exist.

Virtual Function vs Function Overloading

- The prototype for a redefined virtual function must match exactly the prototype specified in the base class.
- This differs from overloading a normal function, in which return types and the number and type of parameters may differ.
- The term *overriding* is used to describe virtual function redefinition by a derived class.

Differences

- When a virtual function is redefined, all aspects of its prototype must be the same.
- If you change the prototype when you attempt to redefine a virtual function, the function will simply be considered overloaded by the C++ compiler, and its virtual nature will be lost.
- Another important restriction is that virtual functions must be non static members of the classes of which they are part.
- They cannot be **friend**s.
- Constructor functions cannot be virtual, but destructor functions can.

Calling a Virtual Function Through a Base Class Reference

- Polymorphic nature of a virtual function is also available when called through a base-class reference.
- A base-class reference can be used to refer to an object of the base class or any object derived from that base.
- When a virtual function is called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.

The Virtual Attribute Is Inherited

- When a virtual function is inherited, its virtual nature is also inherited.
- This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.
- Put differently, no matter how many times a virtual function is inherited, it remains virtual.

Virtual Functions Are Hierarchical

- When a function is declared as **virtual** by a base class, it may be overridden by a derived class.
- However, the function does not have to be overridden.
- When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used.
- Example1:

Virtual Functions Are Hierarchical

- Because inheritance is hierarchical in C++, it makes sense that virtual functions are also hierarchical.
- This means that when a derived class fails to override a virtual function, the first redefinition found in reverse order of derivation is used.
- Example2:

Pure Virtual Functions

- In many situations there can be no meaningful definition of a virtual function within a base class.
- For example, a base class may not be able to define an object sufficiently to allow a base-class virtual function to be created.
- Further, in some situations you will want to ensure that all derived classes override a virtual function.
- To handle these two cases, C++ supports the pure virtual function.

Pure Virtual Functions

- A *pure virtual function* is a virtual function that has no definition within the base class.
- To declare a pure virtual function, use this general form:

  virtual *type func-name(parameter-list*) = 0;

- When a virtual function is made pure, any derived class must provide its own definition.
- If the derived class fails to override the pure virtual function, a compile-time error will result.

Abstract Classes

- A class that contains at least one pure virtual function is said to be *abstract*.
- Because an abstract class contains one or more functions for which there is no definition (that is, a pure virtual function), no objects of an abstract class may be created.
- Instead, an abstract class constitutes an incomplete type that is used as a foundation for derived classes.

Abstract Classes

- Although you cannot create objects of an abstract class, you can create pointers and references to an abstract class.
- This allows abstract classes to support run-time polymorphism, which relies upon base-class pointers and references to select the proper virtual function.

Early vs. Late Binding

- *Early binding* refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time.
- Early binding means that an object and a function call are bound during compilation.)
- Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators.
- The main advantage to early binding is efficiency.
- Because all information necessary to call a function is determined at compile time, these types of function calls are very fast.

Early vs. Late Binding

- The opposite of early binding is *late binding*. As it relates to C++, late binding refers to function calls that are not resolved until run time.
- Virtual functions are used to achieve late binding.
- As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer.
- Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time.
- The main advantage to late binding is flexibility.
- Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code."
- Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times.

Generic Functions(Templates)

- Through a generic function, a single general procedure can be applied to a wide range of data.
- By creating a generic function, you can define the nature of the algorithm, independent of any data.
- Once you have done this, the compiler will automatically generate the correct code for the type of data that is actually used when you execute the function.

Generic Functions

- A generic function is created using the keyword **template**.
- The normal meaning of the word "template" accurately reflects its use in C++.
- It is used to create a template (or framework) that describes what a function will do.

Generic Functions

- The general form of a template function definition is shown here:

  template <class *Ttype*>
  ret-type func-name(*parameter list*)
  {
  // body of function
  }
- Here, *Ttype* is a placeholder name for a data type used by the function.
- This name may be used within the function definition.
- However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function.
- Example: Swap

- Because **swapargs()** is a generic function, the compiler automatically creates three versions of **swapargs()** : one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

Generic Function Restrictions

- Generic functions are similar to overloaded functions except that they are more restrictive.
- When functions are overloaded, you may have different actions performed within the body of each function.
- But a generic function must perform the same general action for all versions—only the type of data can differ.

Generic Classes

- Generic classes are useful when a class uses logic that can be generalized.
- For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information.

Generic Classes

- The general form of a generic class declaration is shown here:

```
template <class Ttype> class class-name {
.
..
}
```

- Here, *Ttype* is the placeholder type name, which will be specified when a class is instantiated.
- If necessary, you can define more than one generic data type using a comma-separated list.

Generic Classes

- Once you have created a generic class, you create a specific instance of that class using the following general form:

  class-name <type> ob;

- Here, *type* is the type name of the data that the class will be operating upon.
- Member functions of a generic class are themselves automatically generic.
- You need not use **template** to explicitly specify them as such.

Example Stack Template

- Two are integer stacks. Two are stacks of
- **doubles**. Pay special attention to these declarations:
- stack<char> s1, s2; // create two character stacks
- stack<double> ds1, ds2; // create two double stacks
- Notice how the desired data type is passed inside the angle brackets. By changing the
- type of data specified when **stack** objects are created, you can change the type of data
- stored in that stack. For example, by using the following declaration, you can create
- another stack that stores character pointers.
- stack<char *> chrptrQ;

Example Stack Template

- You can also create stacks to store data types that you create. For example, if you want to use the following structure to store address information,

```
struct addr {
char name[40];
char street[40];
char city[30];
char state[3];
char zip[12];
};
```

- then to use **stack** to generate a stack that will store objects of type **addr**, use a declaration like this:
- stack<addr> obj;
- An Example with Two Generic Data Types:

The Power of Templates

- Templates help you achieve one of the most elusive goals in programming: the creation of reusable code.
- Through the use of template classes you can create frameworks that can be applied over and over again to a variety of programming situations.
- Once you have written and debugged a template class, you have a solid software component that you can use with confidence in a variety of different situations.

The Power of Templates

- Template functions and classes are already becoming commonplace in programming, and this trend is expected to continue.
- For example, the STL (Standard Template Library) defined by C++ is, as its name implies, built upon templates.

Exception Handling Fundamentals

- C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**.
- In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block.
- If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**).
- The exception is caught, using **catch**, and processed.

Exception Handling Fundamentals

- Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.)
- Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown.

- The general form of **try** and **catch** are shown here.

```
try {
// try block
}
catch (type1 arg) {
// catch block
}
catch (type2 arg) {
// catch block
}
catch (type3 arg) {
// catch block
}
..
.
catch (typeN arg) {
// catch block
}
```

Try

- The **try** can be as short as a few statements within one function or as all encompassing as enclosing the **main()** function code within a **try** block (which effectively causes the entire program to be monitored).

Catch

- When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception.
- There can be more than one **catch** statement associated with a **try**.
- Which **catch** statement is used is determined by the type of the exception.
- That is, if the data type specified by a **catch** matches that of the exception, then that **catch** statement is executed (and all others are bypassed).
- When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create.
- If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.

Throw

- The general form of the **throw** statement is shown here:

  throw *exception*;

- **throw** generates the exception specified by *exception*.
- If this exception is to be caught, then **throw** must be executed either from within a **try** block itself, or from any function called from within the **try** block (directly or indirectly).
- If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination may occur.
- Throwing an unhandled exception causes the standard library function **terminate()** to be invoked.

Example:

```
// A simple exception handling example.
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

Output

- This program displays the following output:

```
Start
Inside try block
Caught an exception -- value is: 100
End
```

Example 2

```
// This example will not work.
#include <iostream>
using namespace std;
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (double i) { // won't work for an int exception
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

Output

- This program produces the following output because the integer exception will not be caught by the **catch(double i)** statement.

```
Start
Inside try block
Abnormal program termination
```

Throwing an exception from a function outside the try block.

```
#include <iostream>
using namespace std;
void Xtest(int test)
{
cout << "Inside Xtest, test is: " << test << "\n";
if(test) throw test;
}
int main()
{
cout << "Start\n";
try { // start a try block
cout << "Inside try block\n";
Xtest(0);
Xtest(1);
Xtest(2);
}
catch (int i) { // catch an error
cout << "Caught an exception -- value is: ";
cout << i << "\n";
}
cout << "End";
return 0;
}
```

Output

- This program produces the following output:

```
Start
Inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught an exception -- value is: 1
End
```

Using Multiple catch Statements

```
void Xhandler(int test)
{
try{
if(test) throw test;
else throw "Value is zero";
}
catch(int i) {
cout << "Caught Exception #: " << i << '\n';
}
catch(const char *str) {
cout << "Caught a string: ";
cout << str << '\n';
}
}
```

```
int main()
{
cout << "Start\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "End";
return 0;
}
```

Output

- This program produces the following output:

```
Start
Caught Exception #: 1
Caught Exception #: 2
Caught a string: Value is zero
Caught Exception #: 3
End
```

Handling Derived-Class Exceptions

```
// Catching derived classes.
#include <iostream>
using namespace std;
class B {
};
class D: public B {
};
int main()
{
D derived;
try {
throw derived;
}
catch(B b) {
cout << "Caught a base class.\n";
}
catch(D d) {
cout << "This won't execute.\n";
}
return 0;
}
```

Catching All Exceptions

- In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type.

```
catch(...) {
// process all exceptions
}
```

Example:

```
try{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}
catch(...) { // catch all exceptions
cout << "Caught One!\n";
}
```

Namespaces

- Purpose is to localize the names of identifiers to avoid name collisions.
- The C++ programming environment has seen an explosion of variable, function, and class names.
- Prior to the invention of namespaces, all of these names competed for slots in the global namespace and many conflicts arose.

Namespaces

- For example, if your program defined a function called **abs()** , it could (depending upon its parameter list) override the standard library function **abs()** 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.
- The situation can be particularly troublesome for class names.

- Prior to **namespace**, the entire C++ library was defined within the global namespace (which was, of course, the only namespace).
- Since the addition of **namespace**, the C++ library is now defined within its own namespace, called **std**, which reduces the chance of name collisions.
- You can also create your own namespaces within your program to localize the visibility of any names that you think may cause conflicts.
- This is especially important if you are creating class or function libraries.

Namespace Fundamentals

- 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** is shown here:

```
namespace name {
// declarations
}
```

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

Example:

```
namespace CounterNameSpace {
    int upperbound;
    int lowerbound;
class counter {
    int count;
    public:
    counter(int n) {
    if(n <= upperbound) count = n;
    else count = upperbound;
    }
void reset(int n) {
    if(n <= upperbound) count = n;
    }
int run() {
    if(count > lowerbound) return count--;
    else return lowerbound;
}
};
}
```

```
int main()
{
CounterNameSpace::upperbound = 100;
CounterNameSpace::lowerbound = 0;
CounterNameSpace::counter ob1(10);
int i;
do {
i = ob1.run();
cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;
CounterNameSpace::counter ob2(20);
do {
i = ob2.run();
cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
cout << endl;
ob2.reset(100);
CounterNameSpace::lowerbound = 90;
do {
i = ob2.run();
cout << i << " ";
} while(i > CounterNameSpace::lowerbound);
return 0;
}
```

using

- Having to specify the namespace and the scope resolution operator each time you need to refer to one quickly becomes a tedious chore.
- The **using** statement was invented to alleviate this problem.
- The **using** statement has these two general forms:
- using namespace *name;*
- using *name*::*member*;

Namespaces

- Namespaces allow to group entities like classes, objects and functions under a name.
- This way the global scope can be divided in "sub-scopes", each one with its own name.
- The format of namespaces is:

```
namespace identifier
{
        entities
}
```

- For example:
- **namespace myNamespace**

```
    {
        int a, b;
    }
```

- In this case, the variables a and b are normal variables declared within a namespace called myNamespace.
- In order to access these variables from outside the myNamespace namespace we have to use the scope operator ::

- // namespaces

```
 #include <iostream>
using namespace std;
namespace first
{ int var = 5; }
namespace second
{ double var = 3.1416; }
 int main ()
 {
cout << first::var << endl;
cout << second::var << endl;
 return 0;
 }
//No redefinition errors happen thanks to namespaces.
```

- // using

```
#include <iostream>
 using namespace std;
namespace first
{ int x = 5; int y = 10; }
namespace second
 { double x = 3.1416; double y = 2.7183; }
int main ()
{
using first::x;
 using second::y;
cout << x << endl; cout << y << endl;
 cout << first::y << endl; cout << second::x << endl;
return 0;
}
```

- The keyword using can also be used as a directive to introduce an entire namespace:
- // using

```
#include <iostream>
using namespace std;
namespace first
{ int x = 5; int y = 10; }
namespace second
{ double x = 3.1416; double y = 2.7183; }
int main ()
{
using namespace first;
cout << x << endl; cout << y << endl;
cout << second::x << endl; cout << second::y << endl;
return 0;
}
```

- using and using namespace have validity only in the same block in which they are stated
- For example, if we had the intention to first use the objects of one namespace and then those of another one, we could do something like:

- // using namespace example

```
 #include <iostream>
using namespace std;
namespace first
{ int x = 5; }
namespace second { double x = 3.1416; }
int main () {
 {
using namespace first;
cout << x << endl;
}
{
using namespace second;
cout << x << endl;
 }
return 0;
}
```

Namespace alias

- We can declare alternate names for existing namespaces according to the following format:

- namespace new_name = current_name;

Namespace std

- All the files in the C++ standard library declare all of its entities within the std namespace.
- That is why we have generally included the using namespace std;

- In the first form, *name* specifies the name of the namespace you want to access.
- All of the members defined within the specified namespace are brought into view (i.e., they

  become part of the current namespace) and may be used without qualification.
- In the second form, only a specific member of the namespace is made visible.

Standard Template Library (*STL*).

- It provides general-purpose, templatized classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks.
- Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.
- At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*.

Containers

- *Containers* are objects that hold other objects, and there are several different types.
- For example, the **vector** class defines a dynamic array, **deque** creates a double-ended
- queue, and **list** provides a linear list.
- These containers are called *sequence containers* because in STL terminology, a sequence is a linear list.

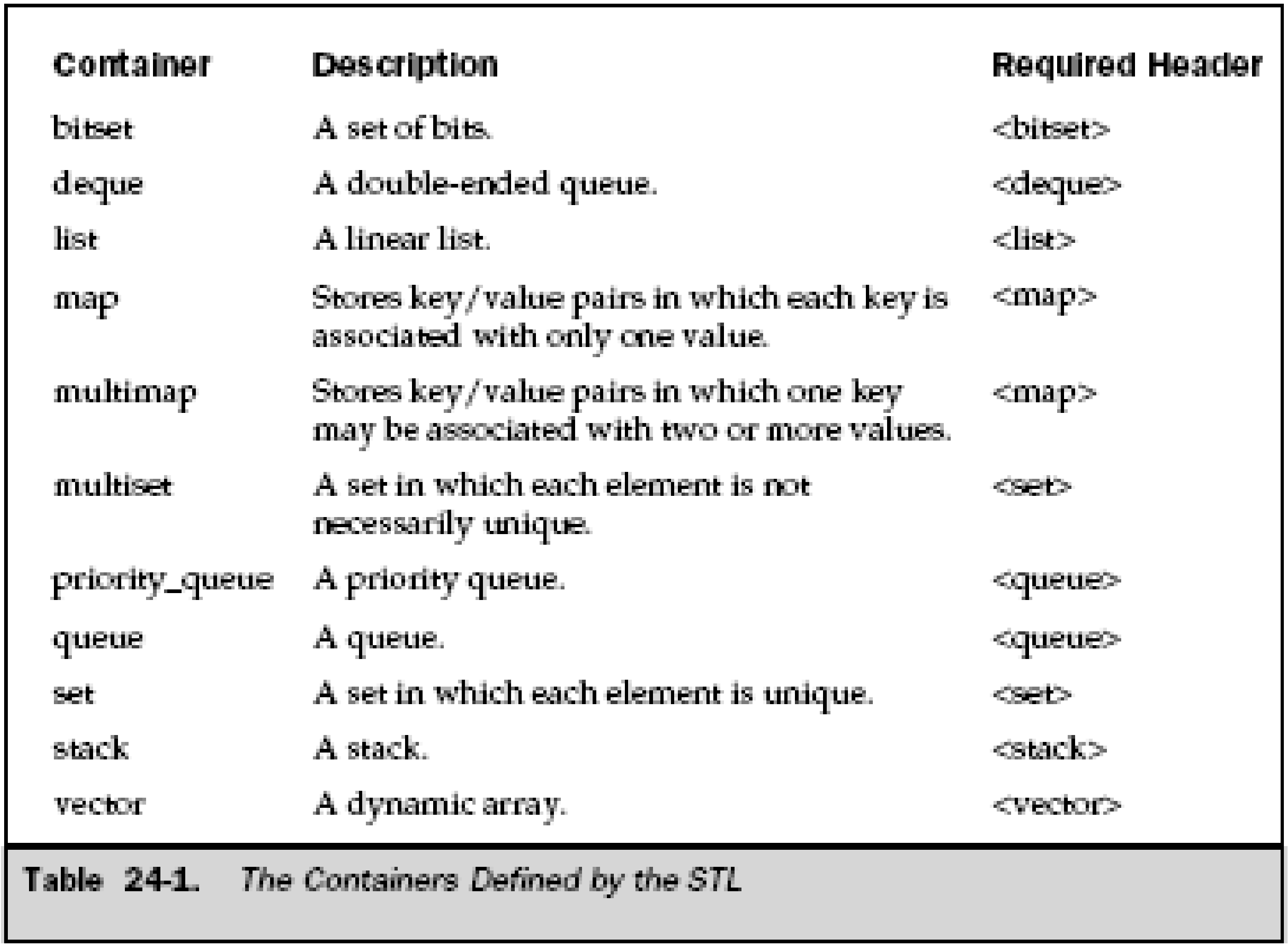
Algorithms

- *Algorithms* act on containers. They provide the means by which you will manipulate the contents of containers.
- Their capabilities include initialization, sorting, searching, and transforming the contents of containers.

Iterators

- *Iterators* are objects that are, more or less, pointers.
- They give you the ability to cycle through the contents of a container in much the same way that you would use a pointer to cycle through an array.

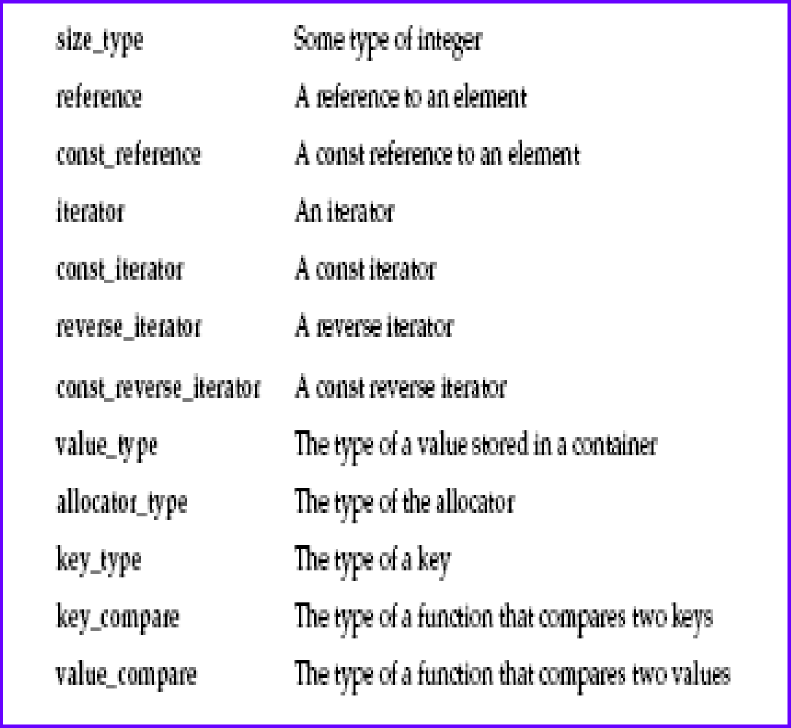
The Container Classes



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 may 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>

Table 24-1. *The Containers Defined by the STL*

- Since the names of the generic placeholder types in a template class declaration are arbitrary, the container classes declare **typedef**ed versions of these types.
- This makes the type names concrete. Some of the most common **typedef** names are shown here:



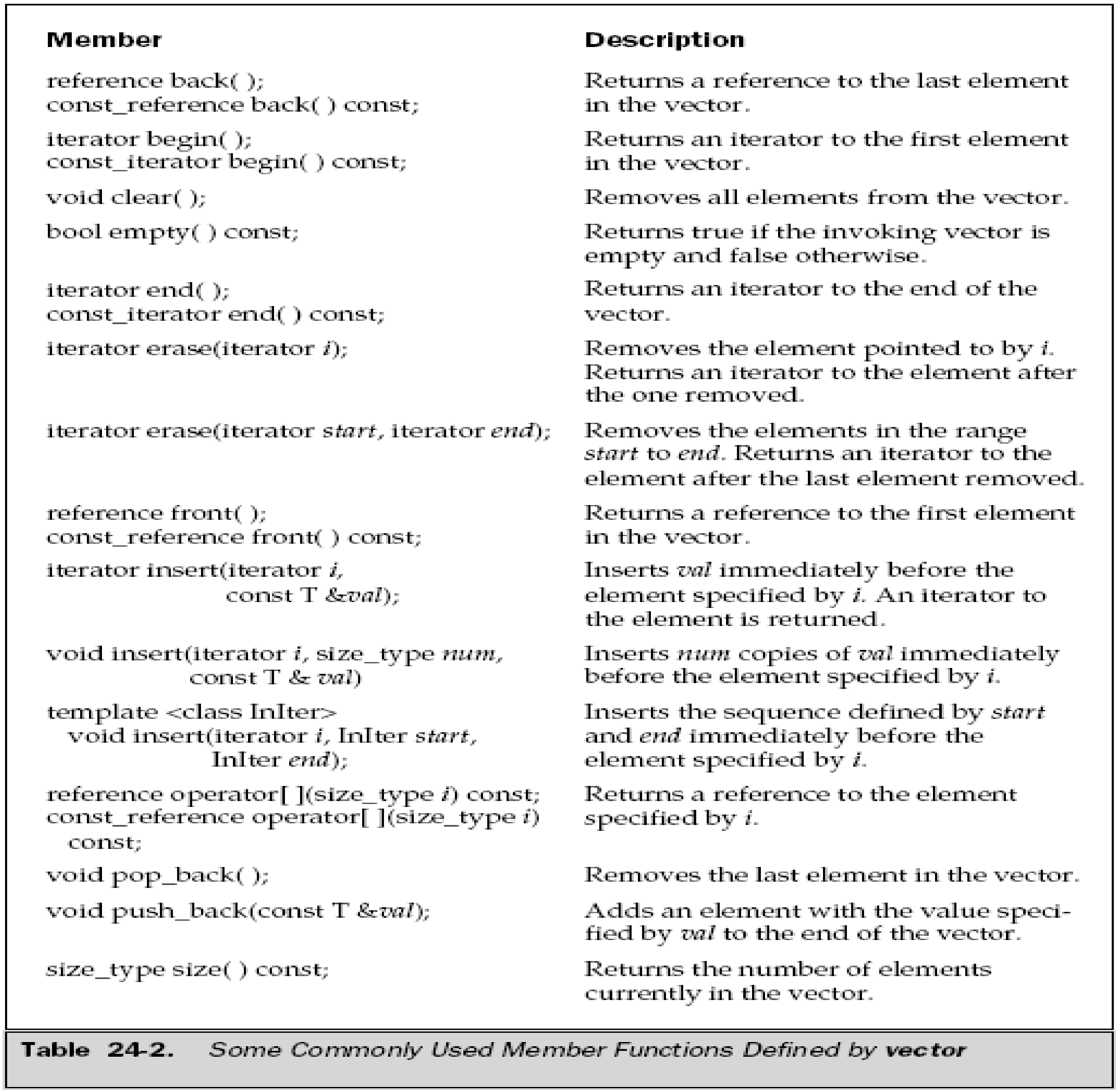
size_type	Some type of integer
reference	A reference to an element
const_reference	A const reference to an element
iterator	An iterator
const_iterator	A const iterator
reverse_iterator	A reverse iterator
const_reverse_iterator	A const reverse iterator
value_type	The type of a value stored in a container
allocator_type	The type of the allocator
key_type	The type of a key
key_compare	The type of a function that compares two keys
value_compare	The type of a function that compares two values

General Theory of Operation

- First, you must decide on the type of container that you wish to use.
- Once you have chosen a container, you will use its member functions to add elements to the container, access or modify those elements, and delete elements.
- One of the most common ways to access the elements within a container is through an iterator.
- The sequence and the associative containers provide the member functions **begin()** and **end()** , which return iterators to the start and end of the container, respectively.

Example: Vectors

- The **vector** class supports a dynamic array.
- This is an array that can grow as needed. As you know, in C++ the size of an array is fixed at compile time.
- While this is by far the most efficient way to implement arrays, it is also the most restrictive because the size of the array cannot be
- adjusted at run time to accommodate changing program conditions.
- A vector solves this problem by allocating memory as needed.



Member	Description

reference back( );
const_reference back( ) const;
	Returns a reference to the last element in the vector.

iterator begin( );
const_iterator begin( ) const;
	Returns an iterator to the first element in the vector.

void clear( );	Removes all elements from the vector.
bool empty( ) const;	Returns true if the invoking vector is empty and false otherwise.

iterator end( );
const_iterator end( ) const;
	Returns an iterator to the end of the vector.

iterator erase(iterator *i*);	Removes the element pointed to by *i*. Returns an iterator to the element after the one removed.
iterator erase(iterator *start*, iterator *end*);	Removes the elements in the range *start* to *end*. Returns an iterator to the element after the last element removed.

reference front( );
const_reference front( ) const;
	Returns a reference to the first element in the vector.

iterator insert(iterator *i*, const T &*val*);	Inserts *val* immediately before the element specified by *i*. An iterator to the element is returned.
void insert(iterator *i*, size_type *num*, const T & *val*)	Inserts *num* copies of *val* immediately before the element specified by *i*.
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*.

reference operator[ ](size_type *i*) const;
const_reference operator[ ](size_type *i*)
const;
	Returns a reference to the element specified by *i*.

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.
size_type size( ) const;	Returns the number of elements currently in the vector.

Table 24-2. *Some Commonly Used Member Functions Defined by* **vector**

Vector

```
// Demonstrate a vector.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;
int main()
{
vector<char> v(10); // create a vector of length 10
int i;
// display original size of v
cout << "Size = " << v.size() << endl;
// assign the elements of the vector some values
for(i=0; i<10; i++) v[i] = i + 'a';
// display contents of vector
cout << "Current Contents:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
cout << "Expanding vector\n";
/* put more values onto the end of the vector,
it will grow as needed */
for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
// display current size of v
cout << "Size now = " << v.size() << endl;
// display contents of vector
cout << "Current contents:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << "\n\n";
// change contents of vector
for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
cout << "Modified Contents:\n";
for(i=0; i<v.size(); i++) cout << v[i] << " ";
cout << endl;
return 0;
}
```

Accessing a Vector Through an Iterator

```
int main()
{
vector<char> v(10); // create a vector of length 10
vector<char>::iterator p; // create an iterator
int i;
// assign elements in vector a value
p = v.begin();
i = 0;
while(p != v.end()) {
*p = i + 'a';
p++;
i++;
}
```

Thank You

Do Practice

All The Best

From : Shraddha Masih