A Very Short Tour of C++
Day 3

Polymorphism

Object-oriented programming languages support *polymorphism* (literal meaning *many shapes*), which is characterized by the phrase "one interface, multiple methods."

In simple terms, polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. A real-world example of polymorphism is a thermostat. No matter what type of furnace your house has (gas, oil, electric, etc.), the thermostat works the same way. In this case, the thermostat (which is the interface) is the same no matter what type of furnace (method) you have. For example, if you want a 70-degree temperature, you set the thermostat to 70 degrees. It doesn't matter what type of furnace actually provides the heat.

This same principle can also apply to programming. For example, you might have a program that defines three different types of stacks. One stack is used for integer values, one for character values, and one for floating-point values. Because of polymorphism, you can define one set of names, **push()** and **pop()** , that can be used for all three stacks. In your program you will create three specific versions of these functions, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being stored. Thus, the interface to a stack—the functions **push()** and **pop()** —are the same no matter which type of stack is being used. The individual versions of these functions define the specific implementations (methods) for each type of data.

Polymorphism helps reduce complexity by allowing the same interface to be used to access a general class of actions. It is the compiler's job to select the *specific action* (i.e., method) as it applies to each situation. You, the programmer, don't need to do this selection manually. You need only remember and utilize the *general interface*.

Among various ways of achieving polymorphism, we have discussed a few: operator overloading, function overloading, virtual function.

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. 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. That is, the redefinition creates a *specific method*.

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. A base-class pointer can be used to point to an object of any class derived from that base. 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. The same effect applies to base-class references.

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

class base {
public:
virtual void vfunc() {
cout << "This is base's vfunc().\n";
}
};

class derived1 : public base {
public:
  void vfunc() {
  cout << "This is derived1's vfunc().\n";
  }
};

class derived2 : public base {
public:
  void vfunc() {
  cout << "This is derived2's vfunc().\n";
  }
};
int main()
{
base *p, b;
derived1 d1;
derived2 d2;

// point to base
p = &b;
p->vfunc(); // access base's vfunc()

// point to derived1
p = &d1;
```

```
p->vfunc(); // access derived1's vfunc()

// point to derived2
p = &d2;
p->vfunc(); // access derived2's vfunc()
return 0;
}
```

This program displays the following:

```
This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().
```

As the program illustrates, inside **base**, the virtual function **vfunc()** is declared. Notice that the keyword **virtual** precedes the rest of the function declaration. When **vfunc()** is redefined by **derived1** and **derived2**, the keyword **virtual** is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

We could have written:

```
d2.vfunc(); // calls derived2's vfunc()
```

Although calling a virtual function in this manner is not wrong, it simply does not take advantage of the virtual nature of **vfunc()** .
In the preceding example, a virtual function was called through a base-class pointer, but the polymorphic nature of a virtual function is also available when called through a base-class reference.
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.
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.

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.

One of the central aspects of object-oriented programming is the principle of "one interface, multiple methods." This means that a general class of actions can be defined, the interface to which is constant, with each derivation defining its own specific operations. In concrete C++ terms, a base class can be used to define the nature of the interface to a general class. Each derived class then implements the specific operations as they relate to the type of data used by the derived type.

One of the most powerful and flexible ways to implement the "one interface, multiple methods" approach is to use virtual functions, abstract classes, and run-time polymorphism. Using these features, you create a class hierarchy that moves from general to specific (base to derived). Following this philosophy, you define all common features and interfaces in a base class. In cases where certain actions can be implemented only by the derived class, use a virtual function. In essence, in the base class you create and define everything you can that relates to the general case. The derived class fills in the specific details.

Template function / class

The template is one of C++'s most sophisticated and high-powered features. Although not part of the original specification for C++, it was added several years ago and is supported by all modern C++ compilers. Using templates, it is possible to create generic functions and classes. In a generic function or class, the type of data upon which the function or class operates is specified as a parameter. Thus, you can use one function or class with several different types of data without having to explicitly recode specific versions for each data type.

A *generic function* defines a general set of operations that will be applied to various types of data. The type of data that the function will operate upon is passed to it as a parameter. Through a generic function, a single general procedure can be applied to a wide range of data. As you probably know, many algorithms are logically the same no matter what type of data is being operated upon. For example, the Quicksort sorting algorithm is the same whether it is applied to an array of integers or an array of floats. It is just that the type of the data being sorted is different. 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. In essence, when you create a generic function you are creating a function that can automatically overload itself.

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, leaving it to the compiler to fill in the details as needed. 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. Although the use of the keyword **class** to specify a generic type in a **template** declaration is traditional, you may also use the keyword **typename**.

The following example creates a generic function that swaps the values of the two variables with which it is called. Because the general process of exchanging two values is independent of the type of the variables, it is a good candidate for being made into a generic function.

```
// Function template example.
#include <iostream>
using namespace std;

// This is a function template.
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

```
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';

cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';

swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars

cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}
```

Let's look closely at this program. The line:

```
template <class X> void swapargs(X &a, X &b)
```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, **X** is a generic type that is used as a placeholder. After the **template** portion, the function **swapargs()** is declared, using **X** as the data type of the values that will be swapped.

In **main()** , the **swapargs()** function is called using three different types of data: **int**s, **double**s, and **char**s. 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.

Here are some important terms related to templates. First, a generic function (that is, a function definition preceded by a **template** statement) is also called a *template function*. When the compiler creates a specific version of this function, it is said to have created a *specialization.* This is also called a *generated function*. The act of generating a function is referred to as *instantiating* it. Put differently, a generated function is a specific instance of a template function. Since C++ does not recognize end-of-line as a statement terminator, the **template** clause of a generic function definition does not have to be on the same line as the function's name. The following example shows another common way to format the swapargs() function.

```
template <class X>
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

If you use this form, it is important to understand that no other statements can occur between the **template** statement and the start of the generic function definition. For example, the fragment shown next will not compile.

```
// This will not compile.
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
```

You can define more than one generic data type in the **template** statement by using a comma-separated list.

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
cout << x << ' ' << y << '\n';
}

int main()
{
myfunc(10, "I like C++");
myfunc(98.6, 19L);
return 0;
}
```

In this example, the placeholder types **type1** and **type2** are replaced by the compiler with the data types **int** and **char ***, and **double** and **long**, respectively, when the compiler generates the specific instances of **myfunc()** within **main()** .
When you create a template function, you are, in essence, allowing the compiler to generate as many different versions of that function as are necessary for handling the various ways that your program calls the function.
Even though a generic function overloads itself as needed, you can explicitly overload one, too. This is formally called *explicit specialization*. If you overload a generic function, that overloaded function overrides (or "hides") the generic function relative to that specific version.

You can mix standard parameters with generic type parameters in a template function. These non-generic parameters work just like they do with any other function.

```
template<class X> void tabOut(X data, int tab)
```

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.
Common examples of the use of generic functions are while carrying out sort, compaction of arrays etc.

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.
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. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.
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. 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.

As you can see, the declaration of a generic class is similar to that of a generic function.

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the **template** specification.

To illustrate the practical benefits of template classes, let's look at one way in which they are commonly applied. We have already discussed about operator overloading, and you know that we can overload the **[ ]** operator. Doing so allows you to create your own array implementations, including "safe arrays" that provide run-time boundary checking. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded **[ ]** subscripting operator, then you can intercept an out-of-range index.

By combining operator overloading with a template class, it is possible to create a generic safe-array type that can be used for creating safe arrays of any data type. This type of array is shown in the following program:

```
// A generic safe array example.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;

template <class AType> class atype
{
AType a[SIZE];
public:
  atype() {
  register int i;
  for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType> AType &atype<AType>::operator[](int i)
{
if(i<0 || i> SIZE-1)
  {
  cout << "\nIndex value of ";
  cout << i << " is out-of-bounds.\n";
  exit(1);
  }
return a[i];
}

int main()
{
atype<int> intob; // integer array
atype<double> doubleob; // double array
int i;

cout << "Integer array: ";
for(i=0; i<SIZE; i++) intob[i] = i;
for(i=0; i<SIZE; i++) cout << intob[i] << " ";
cout << '\n';

cout << "Double array: ";
for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
cout << '\n';

intob[12] = 100; // generates runtime error
return 0;
}
```

This program implements a generic safe-array type and then demonstrates its use by creating an array of **int**s and an array of **double**s. You should try creating other types of arrays. As this example shows, part of the power of generic classes is that they allow you to write the code once, debug it, and then apply it to any type of data without having to re-engineer it for each specific application.

In the template specification for a generic class, you may also specify non-type arguments. That is, in a template specification you can specify what you would normally think of as a standard argument, such as an integer or a pointer. The syntax to accomplish this is essentially the same as for normal function parameters: simply include the type and name of the argument. For example, here is a better way to implement the safe-array class presented in the preceding section.

```
// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, int size is a non-type argument.
template <class AType, int size> class atype
{
AType a[size]; // length of array is passed in size
public:
  atype()
  {
  register int i;
  for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
if(i<0 || i> size-1)
  {
  cout << "\nIndex value of ";
  cout << i << " is out-of-bounds.\n";
  exit(1);
  }

return a[i];
}

int main()
{
atype<int, 10> intob; // integer array of size 10
atype<double, 15> doubleob; // double array of size 15
int i;

cout << "Integer array: ";
for(i=0; i<10; i++) intob[i] = i;
for(i=0; i<10; i++) cout << intob[i] << " ";
cout << '\n';
```

```
cout << "Double array: ";
for(i=0; i<15; i++) doubleob[i] = (double) i/3;
for(i=0; i<15; i++) cout << doubleob[i] << " ";
cout << '\n';

intob[12] = 100; // generates runtime error
return 0;
}
```

Look carefully at the template specification for **atype**. Note that **size** is declared as an **int**. This parameter is then used within **atype** to declare the size of the array **a**. Even though **size** is depicted as a "variable" in the source code, its value is known at compile time. This allows it to be used to set the size of the array. **size** is also used in the bounds checking within the **operator[ ]()** function. Within **main()** , notice how the integer and floating-point arrays are created. The second parameter specifies the size of each array. Non-type parameters are restricted to integers, pointers, or references. Other types, such as **float**, are not allowed. The arguments that you pass to a non-type parameter must consist of either an integer constant, or a pointer or reference to a global function or object. Thus, non-type parameters should themselves be thought of as constants, since their values cannot be changed. For example, inside **operator[ ]()** , the following statement is not allowed.

```
size = 10; // Error
```

Since non-type parameters are treated as constants, they can be used to set the dimension of an array, which is a significant, practical benefit. As the safe-array example illustrates, the use of non-type parameters greatly expands the utility of template classes. Although the information contained in the non-type argument must be known at compile-time, this restriction is mild compared with the power offered by non-type parameters.

A template class can have a default argument associated with a generic type. For example,

```
template <class X=int> class myclass { //...
```

Here, the type **int** will be used if no other type is specified when an object of type **myclass** is instantiated.

It is also permissible for non-type arguments to take default arguments. The default value is used when no explicit value is specified when the class is instantiated. Default arguments for non-type parameters are specified using the same syntax as default arguments for function parameters.

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.
Generic functions and classes provide a powerful tool that you can use to amplify your programming efforts. 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. You are saved from the tedium of creating separate implementations for each data type with which you want the class to work.

While it is true that the template syntax can seem a bit intimidating at first, the rewards are well worth the time it takes to become comfortable with it. 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. One last point: although templates add a layer of abstraction, they still ultimately compile down to the same, high-performance object code that you have come to expect from C++.

Standard Template Library:

Now we will explore what is considered by many to be the most important new feature added to C++ in recent years: the *standard template library* (*STL*). The inclusion of the STL was one of the major efforts that took place during the standardization of C++. 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. 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 nearly any type of data.
The STL is a complex piece of software engineering that uses some of C++'s most sophisticated features. To understand and use the STL, you must have a complete understanding of the C++ language, including pointers, references, and templates. Frankly, the template syntax that describes the STL can seem quite intimidating— although it looks more complicated than it actually is.

At the core of the standard template library are three foundational items: *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 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. In addition to the basic containers, the STL also defines *associative containers,* which allow efficient retrieval of values based on keys. For example, a **map** provides access to values with unique keys. Thus, a **map** stores a key/value pair and allows a value to be retrieved given its key.
Each container class defines a set of functions that may be applied to the container. For example, a list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.
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. Many algorithms operate on a *range* of elements within a container.
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.
There are five types of iterators:

Iterator	Access	Allowed
Random	Access Store and retrieve values.	Elements may be accessed randomly.
Bidirectional	Store and retrieve values.	Forward and backward moving.
Forward	Store and retrieve values.	Forward moving only.
Input	Retrieve, but not store values.	Forward moving only.
Output	Store, but 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. You can increment and decrement them. You can apply the ***** operator to them. Iterators are declared using the **iterator** type defined by the various containers.
The STL also supports *reverse iterators*. Reverse iterators are either bidirectional or random-access iterators that move through a sequence in the 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.
When referring to the various iterator types in template descriptions, the following terms are often used:

Term	Represents
BiIter	Bidirectional iterator
ForIter	Forward iterator
InIter	Input iterator
OutIter	Output iterator
RandIter	Random access iterator

In addition to containers, algorithms, and iterators, the STL relies upon several other standard components for support. Chief among these are allocators, predicates, comparison functions, and function objects.

As explained, containers are the STL objects that actually store data. The containers defined by the STL are shown in the following table. Also shown are the headers necessary 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 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>

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 below:

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

Although the internal operation of the STL is highly sophisticated, to use the STL is actually quite easy. First, you must decide on the type of container that you wish to use. Each offers certain benefits and trade-offs. For example, a **vector** is very good when a random-access, array-like object is required and not too many insertions or deletions are needed. A **list** offers low-cost insertion and deletion but trades away speed. A **map** provides an associative container, but of course incurs additional overhead.

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. Except for **bitset**, a container will automatically grow as needed when elements are added to it and shrink when elements are removed.

Elements can be added to and removed from a container a number of different ways. For example, both the sequence containers (**vector**, **list**, and **deque**) and the associative containers (**map**, **multimap**, **set**, and **multiset**) provide a member function called **insert()** , which inserts elements into a container, and **erase()** , which removes elements from a container. The sequence containers also provide **push_back()** and **push_front()** , which add an element to the end or the beginning of a container, respectively. These functions are probably the most common way that individual elements are added to a sequence container. You can remove individual elements from a sequence container by using **pop_back()** and **pop_front()** , which remove elements from the end and start of the container.

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. These iterators are very useful when accessing the contents of a container. For example, to cycle through a container, you can obtain an iterator to its beginning using **begin()** and then increment that iterator until its value is equal to **end()** .

The associative containers provide the function **find()** , which is used to locate an element in an associative container given its key. Since associative containers link a key with its value, **find()** is how most elements in such a container are located. Since a **vector** is a dynamic array, it also supports the standard array-indexing syntax for accessing its elements.

Once you have a container that holds information, it can be manipulated using one or more algorithms. The algorithms not only allow you to alter the contents of a container in some prescribed fashion, but they also let you transform one type of sequence into another.

An example with Vectors: Perhaps the most general-purpose of the containers is vector. 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. Although a vector is dynamic, you can still use the standard array subscript notation to access its elements.
The template specification for **vector** is shown here:

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

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

```
explicit vector(const Allocator &a = Allocator( ) );
explicit vector(size_type num, const T &val = T ( ),
const Allocator &a = Allocator( ));
vector(const vector<T, Allocator> &ob);
template <class InIter> vector(InIter start, InIter end,
const Allocator &a = Allocator( ));
```

The first form constructs an empty vector. The second form constructs a vector that has *num* elements with the value *val*. The value of *val* may be allowed to default. The third form constructs a vector that contains the same elements as *ob*. The fourth form constructs a vector that contains the elements in the range specified by the iterators *start* and *end*.
Any object that will be stored in a **vector** must define a default constructor. It must also define the < and == operations. Some compilers may require that other comparison operators be defined. (Since implementations vary, consult your compiler's documentation for precise information.) All of the built-in types automatically satisfy these requirements.
Although the template syntax looks rather complex, there is nothing difficult about declaring a vector. Here are some examples:

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

The following comparison operators are defined for **vector**:

```
==, <, <=, !=, >, >=
```

The subscripting operator [ ] is also defined for **vector**. This allows you to access the elements of a vector using standard array subscripting notation.

Here is a short example that illustrates the basic operation of a 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;
}
```

The output of this program is shown here:

```
Size = 10
Current Contents:
a b c d e f g h i j
Expanding vector
Size now = 20
Current contents:
a b c d e f g h i j k l m n o p q r s t
Modified Contents:
A B C D E F G H I J K L M N O P Q R S T
```

Let's look at this program carefully. In **main()** , a character vector called **v** is created with an initial capacity of 10. That is, **v** initially contains 10 elements. This is confirmed by calling the **size()** member function. Next, these 10 elements are initialized to the characters a through j and the contents of **v** are displayed. Notice that the standard array subscripting notation is employed. Next, 10 more elements are added to the end of **v** using the **push_back()** function. This causes **v** to grow in order to accommodate the new elements. As the output shows, its size after these additions is 20. Finally, the values of **v**'s elements are altered using standard subscripting notation.

There is one other point of interest in this program. Notice that the loops that display the contents of **v** use as their target value **v.size()** . One of the advantages that vectors have over arrays is that it is possible to find the current size of a vector. As you can imagine, this can be quite useful in a variety of situations.

As you know, arrays and pointers are tightly linked in C++. An array can be accessed either through subscripting or through a pointer. The parallel to this in the STL is the link between vectors and iterators. You can access the members of a vector using subscripting or through the use of an iterator. The following example shows how.

```
// Access the elements of a vector through an iterator.
#include <iostream>
#include <vector>
#include <cctype>
using namespace std;

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++;
  }

// display contents of vector
cout << "Original contents:\n";
p = v.begin();
while(p != v.end())
  {
  cout << *p << " ";
  p++;
  }
cout << "\n\n";

// change contents of vector
p = v.begin();
while(p != v.end())
  {
  *p = toupper(*p);
  p++;
  }

// display contents of vector
cout << "Modified Contents:\n";
p = v.begin();
while(p != v.end())
  {
```

```
  cout << *p << " ";
  p++;
  }

cout << endl;

return 0;
}
```

The output from this program is

```
Original contents:
a b c d e f g h i j
Modified Contents:
A B C D E F G H I J
```

In the program, 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, you will use a declaration similar to that shown in the example: simply qualify **iterator** with the name of the container. In the program, **p** is initialized to point to the start of the vector by using the **begin()** member function. This function 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. This process is directly parallel to the way a pointer can be used to access the elements of an array. To determine when the end of the vector has been reached, the **end()** member function is employed. This function 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.

An example with strings: Here, we will deal with one of C++'s most important new classes: **string**. The **string** class defines a string data type that allows you to work with character strings much as you do other data types: using operators. The **string** class is closely related to the STL. The C++ string classes make string handling extraordinarily easy. For example, using 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. The following program illustrates these operations.

```
// A short string demonstration.
#include <iostream>
#include <string>
using namespace std;

int main()
{
string str1("Alpha");
string str2("Beta");
string str3("Omega");
string str4;

// assign a string
str4 = str1;
cout << str1 << "\n" << str3 << "\n";

// concatenate two strings
```

```
str4 = str1 + str2;
cout << str4 << "\n";

// concatenate a string with a C-string
str4 = str1 + " to " + str3;
cout << str4 << "\n";

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

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

// create a string object using another string object
string str5(str1);
cout << str5;

// input a string
cout << "Enter a string: ";
cin >> str5;
cout << str5;

return 0;
}
```

This program produces the following output:

```
Alpha
Omega
AlphaBeta
Alpha to Omega
str3 > str1
This is a null-terminated string.
This is a null-terminated string.
Enter a string: STL
STL
```

Notice the ease with which the string handling is accomplished. For example, the **+** is used to concatenate strings and the **>** is used to compare two strings. To accomplish these operations using C-style, null-terminated strings, less convenient calls to the **strcat()** and **strcmp()** functions would be required. Because C++ **string** objects can be freely mixed with C-style null-terminated strings, there is no disadvantage to using them in your program—and there are considerable benefits to be gained.

There is one other thing to notice in the preceding program: the size of the strings is not specified. **string** objects are automatically sized to hold the string that they are given. Thus, when assigning or concatenating strings, the target string will grow as needed to accommodate the size of the new string. It is not possible to overrun the end of the string. This dynamic aspect of **string** objects is one of the ways that they are better than standard null-terminated strings (which *are* subject to boundary overruns)

The STL is now an important, integral part of the C++ language. Many programming tasks can (and will) be framed in terms of it. The STL combines power with flexibility, and while its syntax is a bit complex, its ease of use is remarkable. No C++ programmer can afford to neglect the STL because it will play an important role in the way future programs are written.

Recapitulation of the topics covered on the first two days:
Arguments: In C++, the use of **void** is redundant and unnecessary. As a general rule, in C++ when a function takes no parameters, its parameter list is simply empty; the use of **void** is not required.

Default to int: There has been a fairly recent change to C++ that may affect older C++ code as well as C code being ported to C++. The C language and the original specification for C++ state that when no explicit type is specified in a declaration, type **int** is assumed. However, the "default-to-int" rule was dropped from C++ a couple of years ago, during standardization. The next standard for the C language is also expected to drop this rule, but it is still currently in effect and is used by a large amount of existing code. The "default-to-int" rule is also applied in much older C++ code.

The most common use of the "default-to-int" rule is with function return types. It was common practice to not specify **int** explicitly when a function returned an integer result.

Header files: When C++ was first invented and for several years after that, it used the same style of headers as did C. That is, it used *header files*. In fact, Standard C++ still supports C-style headers for header files that you create and for backward compatibility. However, Standard C++ created a new kind of header that is used by the Standard C++ library. The new-style headers *do not* specify filenames. Instead, they simply specify standard identifiers that may be mapped to files by the compiler, although they need not be. The new-style C++ headers are an abstraction that simply guarantee that the appropriate prototypes and definitions required by the C++ library have been declared.

Since the new-style headers are not filenames, they do not have a **.h** extension. They consist solely of the header name contained between angle brackets. For example, here are some of the new-style headers supported by Standard C++.

```
<iostream> <fstream> <vector> <string>
```

The new-style headers are included using the **#include** statement. The only difference is that the new-style headers do not necessarily represent filenames.

Because C++ includes the entire C function library, it still supports the standard C-style header files associated with that library. That is, header files such as **stdio.h** or **ctype.h** are still available. However, Standard C++ also defines new-style headers that you can use in place of these header files. The C++ versions of the C standard headers simply add a "c" prefix to the filename and drop the **.h**. For example, the C++ new-style header for **math.h** is **<cmath>**. The one for **string.h** is **<cstring>**. Although it is currently permissible to include a C-style header file when using C library functions, this approach is deprecated by Standard C++ (that is, it is not recommended). For this reason, from this point forward, this book will use new-style C++ headers in all **#include** statements. If your compiler does not support new-style headers for the C function library, then simply substitute the old-style, C-like headers.

Namespaces: When you include a new-style header in your program, the contents of that header are contained in the **std** namespace. A *namespace* is simply a declarative region. The purpose

of a namespace is to localize the names of identifiers to avoid name collisions. Elements declared in one namespace are separate from elements declared in another.

Class: In C++, **class** creates a new data type that may be used to create objects of that type. Therefore, an object is an instance of a class in just the same way that some other variable is an instance of the **int** data type, for example. Put differently, a class is a logical abstraction, while an object is real. (That is, an object exists inside the memory of the computer.)
The general form of a simple **class** declaration is

```
class class-name
{
private data and functions

public:
  public data and functions
} object name list;
```

Of course, the *`object name list`* may be empty.

By default, functions and data declared within a class are `private` to that class and may be accessed only by other members of the class. The `public` access specifier allows functions or data to be accessible to other parts of your program. We also looked at the `protected` access specifier that is useful while deriving inherited classes from a base class. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declaration is reached.

Functions that are declared within a class are called *member functions*. Member functions may access any element of the class of which they are a part. This includes all **private** elements. Variables that are elements of a class are called *member variables* or *data members*. Collectively, any element of a class can be referred to as a member of that class.

When it comes time to actually code a function that is the member of a class, you must tell the compiler which class the function belongs to by qualifying its name with the name of the class of which it is a member. The **::** is called the *scope resolution operator*. If we use `void stack::push(int i)`, we are essentially telling the compiler that this version of **push()** belongs to the **stack** class or, put differently, that this **push()** is in **stack'**s scope. In C++, several different classes can use the same function name. The compiler knows which function belongs to which class because of the scope resolution operator.

In general, you should make all data members of a class private to that class. This is part of the way that encapsulation is achieved. However, there may be situations in which you will need to make one or more variables public. (For example, a heavily used variable may need to be accessible globally in order to achieve faster run times.) When a variable is public, it may be accessed directly by any other part of your program. The syntax for accessing a public data member is the same as for calling a member function: Specify the object's name, the dot operator, and the variable name.

Constructors / Destructors: A constructor function is a special function that is a member of a class and has the same name as that class. The general use of a constructor function is for initialization purposes. In C++, constructor functions cannot return values and, thus, have no return type. There are certain default constructors that can be modified to suit one's purpose. It is quite natural to have more than a couple of constructors, including some to which it is possible to pass arguments. Typically, these arguments help initialize an object when it is created. Parameterized constructor functions are very useful because they allow you to avoid having to make an additional function call simply to initialize one or more variables in an object. Each function call you can avoid makes your program more efficient. One of the default constructors, namely, the copy constructor uses object passing by reference.

The complement of the constructor is the *destructor*. In many circumstances, an object will need to perform some action or actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. When an object is destroyed, its destructor (if it has one) is automatically called. There are many reasons why a destructor function may be needed. For example, an object may need to deallocate memory that it had previously allocated or it may need to close a file that it had opened. In C++, it is the destructor function that handles deactivation events. The destructor has the same name as the constructor, but it is preceded by a ~. Note that, like constructor functions, destructor functions do not have return values.

As a general rule, an object's constructor is called when the object comes into existence, and an object's destructor is called when the object is destroyed. Precisely when these events occur is discussed here.

A local object's constructor function is executed when the object's declaration statement is encountered. The destructor functions for local objects are executed in the reverse order of the constructor functions.

Global objects have their constructor functions execute *before* **main()** begins execution. Global constructors are executed in order of their declaration, within the same file. You cannot know the order of execution of global constructors spread among several files. Global destructors execute in reverse order *after* **main()** has terminated.

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

class myclass
{
public:
  int who;
  myclass(int id);
  ~myclass();
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
cout << "Initializing " << id << "\n";
who = id;
}

myclass::~myclass()
{
cout << "Destructing " << who << "\n";
```

```
}
int main()
{
myclass local_ob1(3);
cout << "This will not be first line displayed.\n";
myclass local_ob2(4);
return 0;
}
```

It displays this output:

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1
```

Inline functions: In C++, you can create short functions that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro. To cause a function to be expanded in line rather than called, precede its definition with the inline keyword.

The reason that **inline** functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to **inline** only very small functions. Further, it is also a good idea to **inline** only those functions that will have significant impact on the performance of your program.

Like the **register** specifier, **inline** is actually just a *request*, not a command, to the compiler. The compiler can choose to ignore it. Also, some compilers may not inline all types of functions. For example, it is common for a compiler not to inline a recursive function. You will need to check your compiler's user manual for any restrictions to **inline.** Remember, if a function cannot be inlined, it will simply be called as a normal function.

The `this` *pointer*:

When a member function is called, it is automatically passed an implicit argument that is a pointer to the invoking object (that is, the object on which the function is called). This pointer is called **this**. To understand **this**, first consider a program that creates a class called **pwr** that computes the result of a number raised to some power:

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

```
class pwr
{
double b;
int e;
double val;
public:
  pwr(double base, int exp);
  double get_pwr() { return val; }
};

pwr::pwr(double base, int exp)
{
b = base;
e = exp;
val = 1;
if(exp==0) return;
for( ; exp>0; exp--) val = val * b;
}

int main()
{
pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
cout << x.get_pwr() << " ";
cout << y.get_pwr() << " ";
cout << z.get_pwr() << "\n";
return 0;
}
```

Within a member function, the members of a class can be accessed directly, without any object or class qualification. Thus, inside **pwr(),** the statement

```
b = base;
```

means that the copy of **b** associated with the invoking object will be assigned the value contained in **base**. However, the same statement can also be written like this:

```
this->b = base;
```

The **this** pointer points to the object that invoked **pwr().** Thus, **this –>b** refers to that object's copy of **b**. For example, if **pwr()** had been invoked by **x** (as in **x(4.0, 2)**), then **this** in the preceding statement would have been pointing to **x.** Writing the statement without using **this** is really just shorthand.

Here is the entire **pwr()** function written using the **this** pointer:

```
pwr::pwr(double base, int exp)
{
this->b = base;
this->e = exp;
this->val = 1;
if(exp==0) return;
for( ; exp>0; exp--)
this->val = this->val * this->b;
}
```

Actually, no C++ programmer would write **pwr()** as just shown because nothing is gained, and the standard form is easier. However, the **this** pointer is very important when operators are overloaded and whenever a member function must utilize a pointer to the object that invoked it.

Remember that the **this** pointer is automatically passed to all member functions. Therefore, **get_pwr()** could also be rewritten as shown here:

```
double get_pwr() { return this->val; }
```

In this case, if **get_pwr()** is invoked like this:

```
y.get_pwr();
```

then **this** will point to object **y.**
Two final points about **this:** First, **friend** functions are not members of a class and, therefore, are not passed a **this** pointer. Second, **static** member functions do not have a **this** pointer.

Allocation of memory: `new` and `delete` vs `malloc` and `free`

C++ provides two dynamic allocation operators: **new** and **delete**. These operators are used to allocate and free memory at run time. Dynamic allocation is an important part of almost all real-world programs. C++ also supports dynamic memory allocation functions, called **malloc()** and **free().** These are included for the sake of compatibility with C. However, for C++ code, you should use the **new** and **delete** operators because they have several advantages.

The **new** operator allocates memory and returns a pointer to the start of it. The **delete** operator frees memory previously allocated using **new**. The general forms of **new** and **delete** are shown here:

```
p_var = new type;
delete p_var;
```

Here, *p_var* is a pointer variable that receives a pointer to memory that is large enough to hold an item of type *type*.

Since the heap is finite, it can become exhausted. If there is insufficient available memory to fill an allocation request, then **new** will fail and a **bad_alloc** exception will be generated. This exception is defined in the header **<new>**. Your program should handle this exception and take appropriate action if a failure occurs. If this exception is not handled by your program, then your program will be terminated.

The **delete** operator must be used only with a valid pointer previously allocated by using **new**. Using any other type of pointer with **delete** is undefined and will almost certainly cause serious problems, such as a system crash.

Although **new** and **delete** perform functions similar to **malloc()** and **free(),** they have several advantages. First, **new** automatically allocates enough memory to hold an object of the specified type. You do not need to use the **sizeof** operator. Because the size is computed automatically, it eliminates any possibility for error in this regard. Second, **new** automatically returns a pointer of the specified type. You don't need to use an explicit type cast as you do when allocating memory by using **malloc().** Finally, both **new** and **delete** can be overloaded, allowing you to create customized allocation systems. Although there is no formal rule that states this, it is best not to mix **new** and **delete** with **malloc()** and **free()** in the same program. There is no guarantee that they are mutually compatible.

You can allocate arrays using **new** by using this general form:

p_var = new *array_type [size];*

Here, *size* specifies the number of elements in the array. To free an array, use this form of **delete:**

delete [ ] *p_var*;

Here, the **[ ]** informs **delete** that an array is being released.

You can allocate objects dynamically by using **new**. When you do this, an object is created and a pointer is returned to it. The dynamically created object acts just like any other object. When it is created, its constructor function (if it has one) is called. When the object is freed, its destructor function is executed.

You can allocate arrays of objects, but there is one catch. Since no array allocated by **new** can have an initializer, you must make sure that if the class contains constructor functions, one will be parameterless. If you don't, the C++ compiler will not find a matching constructor when you attempt to allocate the array and will not compile your program.

Function overloading: One way that C++ achieves polymorphism is through the use of 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*.
In general, to overload a function, simply declare different versions of it. The compiler takes care of the rest. You must observe one important restriction when overloading a function: the type and/or number of the parameters of each overloaded function must differ. It is not sufficient for two functions to differ only in their return types. They must differ in the types or number of their parameters. (Return types do not provide sufficient information in all cases for the compiler to decide which function to use.) Of course, overloaded functions *may* differ in their return types, too.
Overloading of constructor function is very common since to overload a constructor is to allow an object to be created by using the most appropriate and natural means for each particular circumstance. Another common reason constructor functions are overloaded is to allow both initialized and uninitialized objects (or, more precisely, default initialized objects) to be created. This is especially important if you want to be able to create dynamic arrays of objects of some class, since it is not possible to initialize a dynamically allocated array. To allow uninitialized arrays of objects along with initialized objects, you must include a constructor that supports initialization and one that does not.

Operator overloading: Polymorphism is also achieved in C++ through operator overloading. As you know, in C++, it is possible to use the << and >> operators to perform console I/O operations. They can perform these extra operations because in the <iostream> header, these operators are overloaded. When an operator is overloaded, it takes on an additional meaning relative to a certain class. However, it still retains all of its old meanings.
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 (for `push` and `pop`, see note related to *stack* below). When an operator is overloaded, none of its original meanings are lost. Instead, the type of objects it can be applied to is expanded.
The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment. After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.
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, however. The way operator functions are written differs between member and nonmember functions.

Memory, heap and stack: The memory a program uses is typically divided into four different areas:
1. The code area, where the compiled program sits in memory.
2. The globals area, where global variables are stored.
3. The heap, where dynamically allocated variables are allocated from.

4. The stack, where parameters and local variables are allocated from.
There isn't really much to say about the first two areas. The heap and the stack are where most of the interesting stuff takes place, and those are the two that will be the focus of this section.
Heap: The heap (also known as the "free store") is a large pool of memory used for dynamic allocation. In C++, when you use the new operator to allocate memory, this memory is assigned from the heap.

```
int *pValue = new int; // pValue is assigned 4 bytes from the heap
int *pArray = new int[10]; // pArray is assigned 40 bytes from the heap
```

Because the precise location of the memory allocated is not known in advance, the memory allocated has to be accessed indirectly — which is why new returns a pointer. You do not have to worry about the mechanics behind the process of how free memory is located and allocated to the user. However, it is worth knowing that sequential memory requests may not result in sequential memory addresses being allocated!
When a dynamically allocated variable is deleted, the memory is "returned" to the heap and can then be reassigned as future allocation requests are received.
The heap has advantages and disadvantages:
1) Allocated memory stays allocated until it is specifically deallocated (beware memory leaks).
2) Dynamically allocated memory must be accessed through a pointer.
3) Because the heap is a big pool of memory, large arrays, structures, or classes should be allocated here

Stack: Consider a bunch of mailboxes, all stacked on top of each other. Each mailbox can only hold one item, and all mailboxes start out empty. Furthermore, each mailbox is nailed to the mailbox below it, so the number of mailboxes cannot be changed. If we can't change the number of mailboxes, how do we get a stack-like behavior, as mentioned below?
1) Look at the top item on the stack (usually done via a function called `top()`)
2) Take the top item off of the stack (done via a function called `pop()`)
3) Put a new item on top of the stack (done via a function called `push()`)

First, we use a marker (like a post-it note) to keep track of where the bottom-most empty mailbox is. In the beginning, this will be the lowest mailbox. When we push an item onto our mailbox stack, we put it in the mailbox that is marked (which is the first empty mailbox), and move the marker up one mailbox. When we pop an item off the stack, we move the marker down one mailbox and remove the item from that mailbox. Anything below the marker is considered "on the stack". Anything at the marker or above the marker is not on the stack.
This is almost exactly analogous to how the call stack works. The call stack is a fixed-size chunk of sequential memory addresses. The mailboxes are memory addresses, and the "items" are pieces of data (typically either variables or addresses). The "marker" is a register (a small piece of memory) in the CPU known as the stack pointer. The stack pointer keeps track of where the top of the stack currently is.
The only difference between our hypothetical mailbox stack and the call stack is that when we pop an item off the call stack, we don't have to erase the memory (the equivalent of emptying the mailbox). We can just leave it to be overwritten by the next item pushed to that piece of memory. Because the stack pointer will be below that memory location, we know that memory location is not on the stack.

So what do we push onto our call stack? Parameters, local variables, and … function calls.
Since parameters and local variables essentially belong to a function, we really only need to consider what happens on the stack when we call a function. Here is the sequence of steps that takes place when a function is called:
1. The address of the instruction beyond the function call is pushed onto the stack. This is how the CPU remembers where to go after the function returns.
2. Room is made on the stack for the function's return type. This is just a placeholder for now.
3. The CPU jumps to the function's code.
4. The current top of the stack is held in a special pointer called the stack frame. Everything added to the stack after this point is considered "local" to the function.
5. All function arguments are placed on the stack.
6. The instructions inside of the function begin executing.
7. Local variables are pushed onto the stack as they are defined.
When the function terminates, the following steps happen:
1. The function's return value is copied into the placeholder that was put on the stack for this purpose.
2. Everything after the stack frame pointer is popped off. This destroys all local variables and arguments.
3. The return value is popped off the stack and is assigned as the value of the function. If the value of the function isn't assigned to anything, no assignment takes place, and the value is lost.
4. The address of the next instruction to execute is popped off the stack, and the CPU resumes execution at that instruction.
Typically, it is not important to know all the details about how the call stack works. However, understanding that functions are effectively pushed on the stack when they are called and popped off when they return gives you the fundamentals needed to understand recursion, as well as some other concepts that are useful when debugging.
The stack has a limited size, and consequently can only hold a limited amount of information. If the program tries to put too much information on the stack, stack overflow will result. *Stack overflow* happens when all the memory in the stack has been allocated — in that case, further allocations begin overflowing into other sections of memory.
Stack overflow is generally the result of allocating too many variables on the stack, and/or making too many nested function calls (where function A calls function B calls function C calls function D etc…) Overflowing the stack generally causes the program to crash.
Here is an example program that causes a stack overflow. You can run it on your system and watch it crash:

```
int main()
{
int nStack[1000000000];

return 0;
}
```

This program tries to allocate a huge array on the stack. Because the stack is not large enough to handle this array, the array allocation overflows into portions of memory the program is not allowed to use. Consequently, the program crashes.
The stack has advantages and disadvantages:
1. Memory allocated on the stack stays in scope as long as it is on the stack. It is destroyed when it is popped off the stack.

2. All memory allocated on the stack is known at compile time. Consequently, this memory can be accessed directly through a variable.
3. Because the stack is relatively small, it is generally not a good idea to do anything that eats up lots of stack space. This includes allocating large arrays, structures, and classes, as well as heavy recursion.

Inheritance: Inheritance is one of the major traits of an object-oriented programming language. In C++, inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific. The process involves first defining a base class, which defines those qualities common to all objects to be derived from the base. The base class represents the most general description. The classes derived from the base are usually referred to as derived classes. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.
The general form for inheritance is

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

Here, *access* is optional. However, if present, it must be **public, private,** or **protected**. For example, using **public** means that all of the public members of the base class will become public members of the derived class. It is important to remember that a derived class has direct access to both its own members and the public members of the base class. A private member of a base class is not accessible by other parts of your program, including any derived class. However, protected members behave differently. If the base class is inherited as public, then the base class' protected members become protected members of the derived class and are, therefore, accessible by the derived class. By using protected, you can create class members that are private to their class but that can still be inherited and accessed by a derived class.
The major advantage of inheritance is that you can create a general classification that can be incorporated into more specific ones. In this way, each object can precisely represent its own subclass. When writing about C++, the terms *base* and *derived* are generally used to describe the inheritance relationship. However, the terms *parent* and *child* are also used. You may also see the terms *superclass* and *subclass*.
Aside from providing the advantages of hierarchical classification, inheritance also provides support for run-time polymorphism through the mechanism of **virtual** functions.

References:

References are perfectly valid types, just like pointers. For instance, just like `int *` is the "pointer to an integer" type, `int &` is the "reference to an integer" type. References can be passed as arguments to functions, returned from functions, and otherwise manipulated just like any other type.

References are just pointers internally; when you declare a reference variable, a pointer to the value being referenced is created, and it's just dereferenced each time the reference variable is used.

Additional information

Friend function / class

It is possible to grant a nonmember function access to the private members of a class by using a **friend**. This is particularly common in operator overloading. A **friend** function has access to all **private** and **protected** members of the class for which it is a **friend**. To declare a **friend** function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

Example:

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

class myclass
{
int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
a = i;
b = j;
}

// Note: sum() is not a member function of any class.
int sum(myclass x)
{
/* Because sum() is a friend of myclass, it can
directly access a and b. */
return x.a + x.b;
}

int main()
{
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

In this example, the **sum()** function is not a member of **myclass**. However, it still has full access to its private members. Also, notice that **sum()** is called without the use of the dot operator. Because it is not a member function, it does not need to be (indeed, it may not be) qualified with an object's name.

Although there is nothing gained by making **sum()** a **friend** rather than amember function of **myclass**, there are some circumstances in which **friend** functions are quite valuable. First, friends can be useful when you are overloading certain types of operators. Second, **friend** functions make the creation of some types of I/O functions easier. The third reason that **friend**

functions may be desirable is that in some cases, two ormore classes may contain members that are interrelated relative to other parts of your program.

It is possible for one class to be a **friend** of another class. When this is the case, the **friend** class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.
#include <iostream>
using namespace std;

class TwoValues
{
int a;
int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;
};

class Min
{
public:
  int min(TwoValues x);
};

int Min::min(TwoValues x)
{
return x.a < x.b ? x.a : x.b;
}

int main()
{
TwoValues ob(10, 20);
Min m;
cout << m.min(ob);
return 0;
}
```

In this example, class **Min** has access to the private variables **a** and **b** declared within the **TwoValues** class.

It is critical to understand that when one class is a **friend** of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the **friend** class.

Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

Nested class

It is possible to define one class within another. Doing so creates a *nested* class. Since a **class** declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class. Frankly, nested classes are seldom used. Because of C++'s flexible and powerful inheritance mechanism, the need for nested classes is virtually nonexistent.

Local classes:

A class may be defined within a function. For example, this is a valid C++ program:

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

void f();

int main()
{
f();
// myclass not known here
return 0;
}

void f()
{
class myclass
{
int i;
public:
  void put_i(int n) { i=n; }
  int get_i() { return i; }
} ob;

ob.put_i(10);
cout << ob.get_i();
}
```

When a class is declared within a function, it is known only to that function and unknown outside of it.

Several restrictions apply to local classes. First, all member functions must be defined within the class declaration. The local class may not use or access local variables of the function in which it is declared (except that a local class has access to **static** local variables declared within the function or those declared as **extern**). It may access type names and enumerators defined by the enclosing function, however. No **static** variables may be declared inside a local class. Because of these restrictions, local classes are not common in C++ programming.

Static:
Both function and data members of a class can be made **static**.

When you precede a member variable's declaration with **static**, you are telling the compiler that only one copy of that variable will exist and that all objects of the class will share that variable. Unlike regular data members, individual copies of a **static** member variable are not made for each object. No matter how many objects of a class are created, only one copy of a **static** data member exists. Thus, all objects of that class use that same variable. All **static** variables are initialized to zero before the first object is created.

When you declare a **static** data member within a class, you are *not* defining it, that is, you are not allocating storage for it. Instead, you must provide a global definition for it elsewhere, outside the class. This is done by re-declaring the **static** variable using the scope resolution operator to identify the class to which it belongs. This causes storage for the variable to be allocated.

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

class shared
{
static int a;
int b;
public:
  void set(int i, int j) {a=i; b=j;}
  void show();
} ;

int shared::a; // define a

void shared::show()
{
cout << "This is static a: " << a;
cout << "\nThis is non-static b: " << b;
cout << "\n";
}

int main()
{
shared x, y;

x.set(1, 1); // set a to 1
x.show();

y.set(2, 2); // change a to 2
y.show();

x.show(); /* Here, a has been changed for both x and y
because a is shared by both objects. */
return 0;
}
```

This program displays the following output when run.

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

Notice that the integer **a** is declared both inside **shared** and outside of it. As mentioned earlier, this is necessary because the declaration of **a** inside **shared** does not allocate storage.

Member functions may also be declared as **static**. There are several restrictions placed on **static** member functions. Actually, static member functions have limited applications, but one good use for them is to "preinitialize" private static data before any object is actually created. Beyond the scope of this class.

File handling:

File handling in C++ works almost identically to terminal input/output. To use files, you write `#include <fstream>` at the top of your source file. Then you can access two classes from the `std` namespace:

- `ifstream` – allows reading input from files
- `ofstream` – allows outputting to files

Each open file is represented by a separate `ifstream` or an `ofstream` object. You can use `ifstream` objects in exactly the same way as `cin` and `ofstream` objects in the same way as `cout`, except that you need to declare new objects and specify what files to open.

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

int main()
{
ifstream source("source-file.txt");
ofstream destination("dest-file.txt");
int x;

source >> x;       // reads one int from source-file.txt
source.close;      // close file as soon as done using it
destination << x; // writes x to dest-file.txt

return 0;
}     // close() called on detination by its destructor
```

Close your files using the `close()` method when you're done using them. This is automatically done for you in the object's destructor, but you often want to close the file ASAP, without waiting for the destructor.

You can specify a second argument to the constructor or the `open` method to specify what "mode" you want to access the file in – read-only, overwrite, write by appending, etc. Check documentation online for details.

Sources:

Paul Kunz's lectures
MIT Introduction to C++
Effective C++, Scott Meyers
C++ The Complete Reference, Herberdt Schildt