C++11

Ratified in 2011, C++11 introduced many changes to C++ standard. Several major features were added in C++11, including both fundamental language constructs (such as lambda expressions) and new libraries (such as one for threading). Although a large number of concepts were added to C++11 this book only uses a handful of them. That being said, it is a worthwhile exercise to peruse additional references to get a sense of all of the additions that were made to the language. This section covers some general C++11 concepts that did not really fit in the other sections of this appendix.

One caveat is that since the C++11 standard is still relatively new, not all compilers are fully C++11-compliant. However, all the C++11 concepts used in this book work in three of the most popular compilers in use today: Microsoft Visual Studio, Clang, and GCC.

It should also be noted that there is a newer version of the C++ standard called C++14. However, C++14 is more of an incremental update, so there are not nearly as many language additions as in C++11. The next major revision to the standard is slated for release in 2017.

`auto`

While the `auto` keyword existed in previous versions of C++, in C++11 it takes on a new meaning. Specifically, this keyword is used in place of a type, and instructs the compiler to deduce the type at compile time. Since the type is deduced at compile time, this means that there is no runtime cost for using `auto`—but it certainly allows for more succinct code to be written.

For example, one headache in old C++ is declaring an iterator (if you are fuzzy on the concept iterators, you can read about them later in this appendix):

```
//Declare a vector of ints
std::vector<int> myVect;
//Declare an iterator referring to begin
std::vector<int>::iterator iter = myVect.begin();
```

However, in C++11 you can replace the complicated type for the iterator with `auto`:

```
//Declare a vector of ints
std::vector<int> myVect;
//Declare an iterator referring to begin (using auto)
auto iter = myVect.begin();
```

Since the return type of `myVect.begin()` is known at compile time, it is possible for the compiler to deduce the appropriate type for `iter`. The `auto` keyword can even be used for basic types such as integers or floats, but the value in these cases is rather questionable. One caveat to keep in mind is that `auto` does not default to a reference nor is it `const`—if these properties are desired, `auto&`, `const auto`, or even `const auto&` can be specified.

nullptr

Prior to C++11, the way a pointer was set to null was either with the number `0` or the macro `NULL` (which is just a `#define` for the number `0`). However, one major issue with this approach is that `0` is first and foremost treated as an integer. This can be a problem in the case of function overloading. For example, suppose the following two functions were defined:

```
void myFunc(int* ptr)
{
   //Do stuff
   //...
}
void myFunc(int a)
{
   //Do stuff
   //...
}
```

An issue comes up if `myFunc` is called with `NULL` passed as the parameter. Although one might expect that the first version would be called, this is not the case. That's because `NULL` is 0, and 0 is treated as an integer. If, on the other hand, `nullptr` is passed as the parameter, it will call the first version, as `nullptr` is treated as a pointer.

Although this example is a bit contrived, the point holds that `nullptr` is strongly typed as a pointer, whereas `NULL` or `0` is not. There's a further benefit that `nullptr` can be easily searched for in a file without any false positives, whereas 0 may appear in many cases where there is not a pointer in use.

References

A **reference** is a variable type that refers to another variable. This means that modifying the reference will modify the original variable. The most basic usage case of references is when writing a function that modifies function parameters. For example, the following function would swap the two parameters `a` and `b`:

```
void swap(int& a, int& b)
{
   int temp = a;
   a = b;
   b = temp;
}
```

Thus if the `swap` function is called on two integer variables, upon completion of the function, these two variables would have their values swapped. This is because `a` and `b` are references to the original variables. Were the `swap` function written in C, we would have to use pointers instead of references. Internally, a reference is in fact implemented as a pointer—however, the semantics

of using a reference are simpler because dereferences are implicit. References are also generally safer to use as function parameters, because it can be assumed that a reference will never be null (though it is technically possible to write malformed code where a reference is null).

Const References

Modifying parameters is only the tip of the iceberg when it comes to references. For nonbasic types (such as classes and structs), passing by reference is almost always going to be more efficient than passing by value. This is because passing by value necessitates creating a copy of the variable—in the case of nonbasic type such as a vector or a string, creating the copy requires a dynamic allocation which adds a huge amount of overhead.

Of course, if the vector or string were just passed into a function by reference, this would mean that the function would be free to modify the original variable. What about the cases where this should be disallowed, such as when the variable is data encapsulated in a class? The solution to this is what's called a **const reference**. A const reference is still passed by reference, but it can only be accessed—no modification is allowed. This is the best of both worlds—a copy is avoided and the function can't modify the data. The following `print` function is one example of passing a parameter by const reference:

```
void print(const std::string& toPrint)
{
   std::cout << toPrint << std::endl;
}
```

In general, for nonbasic types it is a good idea to pass them into functions by const reference, unless the function intends to modify the original variable, in which case a normal reference should be used. However, for basic types (such as integers and floats) it generally is slower to use references as opposed to making a copy. Thus, for basic types it's preferred to pass by value, unless the function intends to modify the original variable, in which case a non-const reference should be used.

Const Member Functions

Member functions and parameters should follow the same rules as standalone functions. So nonbasic types should generally be passed by const reference and basic types should generally be passed by value. It gets a little bit trickier for the return type of so-called getter functions—functions that return encapsulated data. Generally, such functions should return const references to the member data—this is to prevent the caller from violating encapsulation and modifying the data.

However, once const references are being used with classes, it is very important that any member functions that do not modify member data are designated as const member functions. A **const member function** guarantees that the member function in question does not modify internal class data (and it is strictly enforced). This is important because given a const reference

to an object, only const member functions can be called on said object. If you attempt to call a non-const function on a const reference, it causes a compile error.

To designate a member function as const, the `const` keyword appears in the declaration, after the closing parenthesis for the function's parameters. The following `Student` class demonstrates proper usage of references as well as const member functions. Using const appropriately in this manner is often referred to as **const-correctness**.

```
class Student
private:
   std::string mName;
   int mAge;
public:
   Student(const std::string& name, int age)
      : mName(name)
      , mAge(age)
   { }

   const std::string& getName() const {return mName;}
   void setName(const std::string& name) {mName = name;}

   int getAge() const {return mAge;}
   void setAge(int age) {mAge = age;}
};
```

Templates

A **template** is a way to declare a function or class such that it can generically apply to any type. For example, this templated `max` function would support any type that supports the greater than operator:

```
template <typename T>
T max(const T& a, const T& b)
{
   return ((a > b) ? a : b);
}
```

When the compiler sees a call to `max`, it instantiates a version of the template for the type in question. So if there are two calls to `max`–one with integers and one with floats—the compiler would create two corresponding versions of `max`. This means that the executable size and execution performance will be identical to code where two versions of `max` were manually declared.

An approach similar to this can be applied to classes and/or structs, and it is used extensively in STL (covered later in this appendix). However, as with references there are quite a few additional possible uses of templates.

Template Specialization

Suppose there is a templated function called `copyToBuffer` that takes in two parameters: a pointer to the buffer to write to, and the (templated) variable that should be written. One way to write this function might be:

```
template <typename T>
void copyToBuffer(char* buffer, const T& value)
{
   std::memcpy(buffer, &value, sizeof(T));
}
```

However, there is a fundamental problem with this function. While it'll work perfectly fine for basic types, nonbasic types such as string will not function properly. This is because the function will perform a shallow copy as opposed to a deep copy of the underlying data. To solve this issue, a specialized version of `copyToBuffer` can be created that performs the deep copy for strings:

```
template <>
void copyToBuffer<std::string>(char* buffer, const std::string& value)
{
   std::memcpy(buffer, value.c_str(), value.length());
}
```

Then, when `copyToBuffer` is invoked in code, if the type of value is a `string` it will choose the specialized version. This sort of specialization can also be applied to a template that takes in multiple template parameters—in which case it is possible to specialize on any number of the template parameters.

Static Assertions and Type Traits

Runtime assertions are very useful for validation of values. In games, assertions are often preferred over exceptions both because there is less overhead and the assertions can be easily removed for an optimized release build.

A **static assertion** is a type of assertion that is performed at compile time. Since this assertion is during compilation, the Boolean expression to be validated must also be known at compilation. Here's a very simple example of a function which will not compile due to the static assertion:

```
void test()
{
   static_assert(false, "Doesn't compile!");
}
```

Of course, a static assert with a `false` condition doesn't really accomplish much other than halt compilation. An actual usage case is combining static assertions with the C++11 `type_traits`

header in order to disallow templated functions on certain types. Returning to the earlier `copyToBuffer` example, it would be preferable if the generic version of the function only worked on basic types. This could be accomplished with a static assertion, like so:

```
template <typename T>
void copyToBuffer(char* buffer, const T& value)
{
   static_assert(std::is_fundamental<T>::value,
      "copyToBuffer requires specialization for non-basic types.");
   std::memcpy(buffer, &value, sizeof(T));
}
```

The `is_fundamental` value will only be true in the case where `T` is a basic type. This means that any calls to the generic version of `copyToBuffer` will not compile if `T` is nonbasic. Where this gets interesting is when specializations are thrown into the mix—if the type in question has a template specialization associated with it, then the generic version is ignored, thus skipping the static assertion. This means that if the string version of `copyToBuffer` were still written as in that earlier example, calls to the function with a string as the second parameter would work just fine.

Smart Pointers

A **pointer** is a type of variable that stores a memory address, and is a fundamental construct used by C/C++ programmers. However, there are a few common issues that can crop up when using pointers incorrectly. One such issue is a memory leak—when memory is dynamically allocated on the heap, but never deleted. For example, the following class leaks memory:

```
class Texture
{
private:
   struct ImageData
   {
      //...
   };
   ImageData* mData;
public:
   Texture(const char* filename)
   {
      mData = new ImageData;
      //Load ImageData from the file
      //...
   }
};
```

Notice how there is memory dynamically allocated in the constructor of the class, but that memory is not deleted in the destructor. To fix this memory leak, we need to add a destructor that deletes `mData`. The corrected version of `Texture` follows:

```
class Texture
{
private:
   struct ImageData
   {
      //...
   };
   ImageData* mData;
public:
   Texture(const char* fileName)
   {
      mData = new ImageData;
      //Load ImageData from the file
      //...
   }
   ~Texture()
   {
      delete mData; //Fix memory leak
   }
};
```

A second, more insidious, issue can crop up when multiple objects have pointers to the same variable that was dynamically allocated. For example, suppose there is the following `Button` class (that uses the previously declared `Texture` class):

```
class Button
{
private:
   Texture* mTexture;
public:
   Button(Texture* texture)
      : mTexture(texture)
   {}
   ~Button()
   {
      delete mTexture;
   }
};
```

The idea is that each `Button` should display a `Texture`, and the `Texture` must have been dynamically allocated beforehand. However, what happens if two instances of `Button` are created, both pointing to the same `Texture`? As long as both buttons are active, everything will work fine. But once the first `Button` instance is destructed, the `Texture` will no longer be valid. But the second `Button` instance would still have a pointer to that newly deleted `Texture`, which in the best case causes some graphical corruption, and in the worst case causes the program to crash. This issue is not easily solvable with normal pointers.

Smart pointers are a way to solve both of these issues, and as of C++11 they are now part of the standard library (in the `memory` header file).

Shared Pointers

A **shared pointer** is a type of smart pointer that allows for multiple pointers to the same dynamically allocated variable. Behind the scenes, a shared pointer tracks the number of pointers to the underlying variable, which is a process called **reference counting**. The underlying variable is only deleted once the reference count hits zero. In this way, a shared pointer can ensure that a variable that's still being pointed at is not deleted prematurely.

To construct a shared pointer, it is preferred to use the `make_shared` templated function. Here's a simple example of using shared pointers:

```
{
   //Construct a shared pointer to an int
   //Initialize underlying variable to 50
   //Reference count is 1
   std::shared_ptr<int> p1 = std::make_shared<int>(50);
   {
      //Make a new shared pointer that's set to the
      //same underlying variable.
      //Reference count is now 2
      std::shared_ptr<int> p2 = p1;

      //Dereference a shared_ptr just like a regular one
      *p2 = 100;
      std::cout << *p2 << std::endl;
   } //p2 destructed, reference count now 1
} //p1 destructed, reference count 0, so underlying variable is deleted
```

Note that both the `shared_ptr` itself and the `make_shared` function are templated by the type of the underlying dynamically allocated variable. The `make_shared` function automatically performs the actual dynamic allocation—notice how there are no direct calls to either `new` or `delete` in this code. It is possible to directly pass a memory address into the constructor of a `shared_ptr`, but this approach is not recommended unless absolutely necessary, as it is less efficient and more error-prone than using `make_shared`.

If you want to pass a shared pointer as a parameter to a function, it should always be passed by value, as if it were a basic type. This is contrary to the usual rules of passing by reference, but it is the only way to ensure the reference count of the shared pointer is correct.

Putting this all together, it means that the `Button` class from earlier in this section could be rewritten to instead use a `shared_ptr` to a `Texture`, as shown in the following code. In this way, the underlying `Texture` data is guaranteed to never be deleted as long as there are active shared pointers to that `Texture`.

```
class Button
{
private:
   std::shared_ptr<Texture> mTexture;
public:
   Button(std::shared_ptr<Texture> texture)
      : mTexture(texture)
   {}
   //No destructor needed, b/c smart pointer!
};
```

There's another related feature of `shared_ptr` that bears mentioning. If a class needs to get a `shared_ptr` to itself, it should not manually construct a new `shared_ptr` from the `this` pointer, as this would not take into account any existing references. Instead, there is a template class you can inherit from called `enable_shared_from_this`. For example, if `Texture` needs to be able to get a `shared_ptr` to itself, it could inherit from `enable_shared_from_this` as follows:

```
class Texture: public std::enable_shared_from_this<Texture>
{
   //Implementation
   //...
};
```

Then, inside any of `Texture`'s member functions, you can call the `shared_from_this` member function, which will return a `shared_ptr` with the correct reference count.

There also are templated functions that can be used to cast between shared pointers to different classes in a hierarchy: `static_pointer_cast` and `dynamic_pointer_cast`.

Unique Pointer

A **unique pointer** is similar to a shared pointer, except it guarantees that only one pointer can ever point to the underlying variable. If you try to assign one unique pointer to another, it results in an error. This means that unique pointers don't need to track a reference count—they simply automatically delete the underlying variable when the unique pointer is destructed.

For unique pointers, use `unique_ptr` and `make_unique`–beyond the lack of reference counting, the code for using `unique_ptr` is very similar to code for using `shared_ptr`.

Weak Pointer

Behind the scenes, a `shared_ptr` actually has two types of reference counts: a strong reference count and a weak reference count. When the strong reference count hits zero, the underlying object is destroyed. However, the weak reference count has no bearing on whether or not the underlying object is destroyed. This leads to a weak pointer, which holds a weak

reference to the object controlled by a shared pointer. The basic idea of a weak pointer is it allows code that doesn't actually want to own an object to safely check whether or not said object still exists. The class used for this in C++11 is `weak_ptr`.

Suppose `sp` is already declared as a `shared_ptr<int>`. You could then create a `weak_ptr` directly from the `shared_ptr` as follows:

```
std::weak_ptr<int> wp = sp;
```

You can then use the expired function to test whether or not the weak pointer still exists. And if it's not expired, you can use lock to reacquire a `shared_ptr`, which will increase the strong reference count. This would look like:

```
if (!wp.expired())
{
   //This will increase the strong reference count
   std::shared_ptr<int> sp2 = wp.lock();
   //Now use sp2 like a shared_ptr
   //...
}
```

Weak pointers can also be used to avoid a circular reference. Specifically, if object A has a `shared_ptr` to object B, and object B has a `shared_ptr` to object A, there is no way object A or B can ever be deleted. However, if one of them has a `weak_ptr`, then the circular reference is avoided.

Caveats

There are a couple of things to watch out for with regards to smart pointers as implemented in C++11. First of all, they are difficult to use correctly with dynamically allocated arrays. If you want to use a smart pointer to an array, it is generally simpler to use an STL container array. It should also be noted that in comparison to normal pointers, smart pointers do come with a slight added memory overhead and performance cost. So for code that needs to be absolutely as fast as possible, it is not wise to use smart pointers. But for most typical usage cases, it's safer and easier (and thus, likely preferred) to use smart pointers.

STL Containers

The C++ **standard template library** (**STL**) contains a large number of container data structures. This section summarizes the most commonly used containers and their typical usage cases. Each container is declared in a header file corresponding to the container name, so it's not uncommon to need to include several headers to support several containers.

array

The `array` container (added in C++11) is essentially a wrapper for a constant size array. Because it is constant size, there are no `push_back` member functions or the like. Indices into the `array` can be accessed using the standard array subscript operator `[]`. Recall that the main advantage of arrays (in general) is that random access can be performed with an algorithmic complexity of *O*(1).

While C-style arrays serve the same purpose, one advantage of using the `array` container is that it supports iterator semantics. Furthermore, it is possible to employ bounds checking if the `at` member function is used instead of the array subscript operator.

vector

The `vector` container is a dynamically sized array. Elements can be added and removed from the back of a vector using `push_back` and `pop_back`, with *O*(1) algorithmic complexity. It is also possible to use `insert` and `remove` at any arbitrary location in the vector. However, these operations require copying some or all of the data in the array, which can make them computationally expensive. Resizing a vector is expensive for the same reason, in spite of its *O*(*n*) algorithmic complexity. This further means that even though `push_back` is considered *O*(1), calling it on a full vector will incur copying costs. As with `array`, bounds checking is performed if the `at` member function is used.

If you know how many elements you need to place in the vector, you can use the `reserve` member function to allocate space to fit that many elements. This will avoid any cost of growing and copying the vector as you add elements, and can save a tremendous amount of time.

For adding elements to a vector, C++11 provides a new member function `emplace_back`. The difference between `emplace_back` and `push_back` is apparent when you have a vector of a nonbasic type. Suppose you have a vector of a custom class `Student`. Suppose that the constructor of Student takes in the name of the student and their grade. If you were to use `push_back`, you might write code like this:

```
students.push_back(Student("John", 100));
```

This code first constructs a temporary instance of the Student class, and then makes a copy of this temporary instance in order to add it to the vector. However, `emplace_back` can construct the object in place, which avoids creating a temporary. You would call `emplace_back` as follows:

```
students.emplace_back("John", 100);
```

Notice how the call to `emplace_back` does not explicitly mention the `Student` type. This is called **perfect forwarding**, because the parameters are forwarded to the `Student` that is constructed in the vector.

There is no disadvantage of using `emplace_back` in lieu of `push_back`. All the other STL containers (other than array) support emplace functionality, as well, so you should get into the habit of using emplace functions to add elements to containers.

`list`

The `list` container is a doubly linked list. Elements can be added/removed from the front and back with guaranteed *O*(1) algorithmic complexity. Furthermore, given an iterator at an arbitrary location in the list, it is possible to `insert` and `remove` with *O*(1) complexity. Recall that lists do not support random access of specific elements. One advantage of a linked list is that it can never really be "full"—elements are added one at a time, so there is no need to worry about resizing a linked list. However, it should be noted that one disadvantage of a linked list is that because elements are not next to each other in memory, they are not as cache-friendly as an array. It turns out that cache performance is actually a significant bottleneck on modern computers. So as long as the size of each element is relatively small (64 bytes or less), a vector will almost always outperform a list.

`forward_list`

The `forward_list` container (added in C++11) is a singly linked list. This means that `forward_list` only supports *O*(1) addition and removal from the front of the list. The advantage of this is that it uses less memory per node in the list.

`map`

A `map` is an ordered container of {key, value} pairs, that are ordered by the key. Each key in the map must be unique and support **strict weak ordering**, meaning that if key A is less than B, then key B cannot be less than or equal to A. If you wish to use a custom type as a key, typically you override the less than operator. A map is implemented as a type of binary search tree, which means that lookup by key has an average algorithmic complexity of *O*(log(*n*)). Since it is ordered, iterating through the map is guaranteed to be sorted in ascending order.

`set`

A `set` is like map, except there is no pair—the key is simply also the value. All other behavior is identical.

`unordered_map`

The `unordered_map` container (added in C++11) is a hash table of {key, value} pairs. Each key must be unique. Since it is a hash table, lookup can be performed with an algorithmic complexity of *O*(1). However, it's unordered which means iterating through an `unordered_map` will not yield any meaningful order. Similarly, there is a hash set container called

`unordered_set`. For both `unordered_map` and `unordered_set`, hashing functions are provided for built-in types. If you wish to hash a custom type, you must provide your own specialization of the templated `std::hash` function.

Iterators

An **iterator** is a type of object whose intent is to allow for traversal through a container. All STL containers support iterators, and this section covers the common usage cases.

The following code snippet constructs a vector, adds the first five Fibonacci numbers to the vector, and then uses iterators to print out each element in the vector:

```
std::vector<int> myVec;
myVec.emplace_back(1);
myVec.emplace_back(1);
myVec.emplace_back(2);
myVec.emplace_back(3 );
myVec.emplace_back(5);

//Iterate through vector, and output each element
for(auto iter = myVec.begin();
    iter != myVec.end();
    ++iter)
{
   std::cout << *iter << std::endl;
}
```

To grab an iterator to the first element in an STL container, the `begin` member function is used, and likewise the `end` member function grabs an iterator to one past the last element. Notice that the code used `auto` to declare the type of the iterator, in order to avoid needing to spell out the full type (which in this case is `std::vector<int>::iterator`).

Also notice that the iterator is incremented to the next element by using the prefix `++` operator—for performance reasons, the prefix operator should be used in lieu of the postfix operator. Finally, iterators are dereferenced like pointers are dereferenced—this is how the underlying data at the element is accessed. This can be tricky if the underlying element is a pointer, because there are two dereferences: first of the iterator and then of the pointer itself.

All STL containers also support two kinds of iterators: the normal iterator as shown earlier, and a `const_iterator`. The difference is that a `const_iterator` does not allow modification of the data in the container, whereas a normal iterator does. This means that if code has a const reference to an STL container, it is only allowed to use a `const_iterator`.

Range-Based For Loop

In the case where it is simply desired to loop through an entire container, it is simpler to use a new C++11 addition called the **range-based for loop**. The loop just mentioned could instead be rewritten as follows:

```
//Iterate using a range-based for
for (auto i : myVec)
{
   std::cout << i << std::endl;
}
```

A range-based for loop looks much like what a `foreach` might look like in other languages such as Java or C#. This code grabs each element in the container, and saves it into the temporary variable `i`. The loop ends only once all elements have been visited. In a range-based for, it is possible to grab each element by value or by reference. This means that if it is desired to modify elements in the container, references should be used, and furthermore for nonbasic types either references or const references should always be used.

Internally, a range-based for will work on any container that supports STL-style iterator semantics (e.g., there is an `iterator` member, a `begin`, an `end`, the iterator can be incremented, dereferenced, and so on). This means that it is possible to create a custom container that supports the range-based for loop.

Other Uses of Iterators

There are a multitude of functions in the `algorithm` header that use iterators in one way or another. However, one other common use of iterators is the `find` member function that `map`, `set`, and `unordered_map` support. The `find` member function searches through the container for the specified key, and returns an iterator to the corresponding element in the container. If the key is not found, `find` will instead return an iterator equal to the `end` iterator.

Additional Readings

Meyers, Scott. (2014, December). *Effective Modern C++*. O'Reilly Media.

Stroustrup, Bjarne. (2013, May). *The C++ Programming Language, 4th ed*. Addison-Wesley.