A Short Introduction to C++ for Graphics Programmers (who know Java and C)

This is a brief introduction to C++, for people who know Java and C. We will focus particularly on aspects of C++ that will be the most useful for graphics programming, and will intentionally avoid some aspects of C++, which, while important, are not essential for writing programs needed in this course. **Disclaimer:** This information has all be hastily thrown together. The code fragments have not been tested. I apologize in advance for any misinformation.

**Primitive types and objects:** In Java, everything is either a primitive type (int, char, float, etc.) or an object. Objects in Java are essentially references to objects. C++ inherits its basic types from C, and adds an two additional types, classes and references. The C++ types include primitive types (int, char, float etc.), enumerations, C-style structures and classes, pointers, and references.

As in C, a pointer is an address in memory, and an array is a pointer to the first element of an array. References in C++ are similar to references in Java, but they can be used for primitive types as well as for objects. Their most common use is in passing parameters to functions. Note that structures and classes in C++ behave differently than in Java. In Java, assigning one class object to another copies the reference. In C++, assigning one class object (by default) does a byte-by-byte copy of one object to the other.

**Constants and Enumerations:** One interesting feature of C++ is the ability to declare that an object is constant. This means that, once initialized, its value cannot be changed. (This avoids the `#define` constants that crop up in many C programs.) Here is an example:

```cpp
const float FREEZING_POINT = 32.0;
const int WINDOW_WIDTH = 800;
const int WINDOW_HEIGHT = 300;
const char QUIT = 'q';
```

By convention, constants are expressed using all capital letters.

A common use for constants is for indicating quantities that take on a discrete set of values. As in C, C++ offers enumerations, which provide an easy way to define such constants. Here are some examples:

```cpp
enum DayOfWeek { SUN, MON, TUE, WED, THU, FRI, SAT }
enum ExecutionMode { DEBUG, PERF_TEST, RELEASE };
```

```cpp
DayOfWeek today = TUE;
ExecutionMode execMode = DEBUG;
```

**Classes:** Class syntax in C++ is quite similar to Java. Note however, that a semicolon is placed at the end of a C++ class. Unlike Java, it is possible to define class methods outside the class body as well as inside. (In C++, class methods are usually called member functions.)

```cpp
// This would appear in file Vector2d.h
class Vector2d {
    public:
        Vector2d() { x = y = 0; } // default constructor
        Vector2d(double xx, double yy); // constructor
        double getX(); { return x; } // getters
        double getY(); { return y; }
        void setX (double xx) { x = xx; } // setters
        void setY (double yy) { y = yy; }
```
Vector2d addTo(Vector2d v); // add to vector v
private:
  double x;       // member data
  double y;
};

We have defined the getters and setters inside the class. We have chosen to define the constructor and the addTo function outside the class (see below).

//----------------------------------------------------------
// This would appear in file Vector2d.cpp
//----------------------------------------------------------
#include "Vector2d.h"
Vector2d::Vector2d(double xx, double yy)
  { x = xx; y = yy; }
Vector2d Vector2d::addTo(Vector2d v)
  { x += v.x; y += v.y; }

Notice that, when outside the class, it is necessary to use the scope resolution operator, “::”, to indicate that a name is associated with a particular class. Thus, when we define the function addTo outside the class, we need to specify Vector2d::addTo, so the compiler knows we are talking about a member of Vector2d.

What is the difference between defining a method inside or outside the class body? Some C++ purists insist that all member functions be defined outside the class (to prevent a user of the class from seeing them). Other C++ programmers define short 1-line member functions inside the class definitions (e.g., getters and setters) and define the rest externally. C++ takes a definition inside the class body to be a hint that this should be an inline function, which means that the function is expanded, rather than being called. This produces more efficient code, but excessive use of this feature results in unnecessarily long executable files (so called, “code bloat”).

Stream I/O: Input and output in C++ is performed by the operators >> and <<, respectively. The standard output stream is called “cout” and the standard stream is called “cin”. Here is a simple example.

    int x, y;
    cin >> x >> y; // input x and y
    cout << "The value of x is " << x << " and y is " << y << "\n";

The character “\n” generates an end-of-line. It is also possible to use standard C I/O (printf and scanf), but it is not a good idea to mix C++ stream I/O with C standard I/O in the same program.

Include files and namespaces: Following C’s convention, declarations are stored in files ending in “.h” and most code is stored in files ending in “.cpp” or “.cc”. Objects like cin and cout are defined by the system. At the start of each program, it is common to begin with a number of directives that include common system declarations. Here are some of the most useful ones.

    #include <cstdlib> // standard definitions from C (such as NULL)
    #include <cstdio> // standard I/O for C-style I/O (scanf, printf)
    #include <cmath> // standard C math definitions (sqrt, sin, cos, etc.)
    #include <iostream> // C++ stream I/O
    #include <string> // C++ string manipulation
    #include <vector> // C++ STL vector (an expandable vector object)
    #include <list> // C++ STL list (a linked list)

In order to keep your program names from clashing with C++ program objects, many named entities are organized into namespaces. Most system objects are stored in a namespace called “std”. This includes, for example, cin and cout, mentioned above. To access objects from this namespace, you need to use a scope
resolution operator, "::". For example, to refer to cin, you would use std::cin and the refer to cout you would use std::cout. To avoid this extra verbiage, you can invoke the "using" command to provide direct access to these names.

```cpp
using std::cin; // make std::cin accessible
using std::cout; // make std::cout accessible
using namespace std; // make all of std accessible
```

**Memory Allocation and Deallocation:** One of the principal differences with C++ and Java is the need to explicitly deallocate memory that has been allocated. Failure to deallocate memory that has been allocated results in a memory leak, which if not handled, can cause your program to exhaust all its available memory prematurely and crash. As in Java, memory is allocated using new. This returns a pointer to the newly allocated object. Unlike the primitive C function malloc, the new operator returns an object of the specified type, and performs initialization by invoking the constructor. For example, to allocate an object of type Vector2d, we could do the following.

```cpp
Vector2d* p; // p is a pointer to a Vector2d
p = new Vector2d(3, -4); // allocate a vector, initialized to (3, -4)
p->setY(2.6); // set its y-coordinate to 2.6
cout << p->getX(); // print its x-coordinate
delete p; // deallocate p’s memory
```

Recall from C that, when dealing with pointers, the "*" operator is used to dereference its value and if `p` is a pointer to a structure or class, then `p->xxx` is used to access member `xxx`.

**Array Allocation:** It is also possible to use new and delete to allocate arrays. This is a common way to generate vectors whose size is known only at execution time. When deleting such an array, use "delete []". Here is any example.

```cpp
int n = 100;
Vector2d* p = new Vector2d[n]; // allocate an array of n vectors
delete [] p;
```

**Constructors and Destructors:** The most common place where memory is allocated and deallocated is when classes are first constructed or destroyed, or in class member functions that insert new entries into a dynamic object. If a class allocates memory, it is important that when the class object ceases to exist, it must deallocate all the memory that it allocated. Whenever an object is about to cease to exist (e.g., the scope in which it was defined is exiting), the system automatically invokes a special class function called a destructor. Given a class `X`, the corresponding destructor is named `~X`. Here is a simple example, of a class, which allocates an array.

```cpp
class VectorArray {
public:
    VectorArray(int capac); // constructor
    Vector2d at(int i) { return A[i]; } // some functions omitted...
    ~VectorArray(); // destructor
private:
    int n; // array capacity
    Vector2d* A; // array storage
}; // constructor
VectorArray::VectorArray(int capac) {
    n = capac;
    A = new Vector2d[n]; // allocate array storage
```
VectorArray::~VectorArray() {
    delete[] A; // deallocate array storage
}

Note that you do not invoke the destructor (in fact, you can’t). The system does it automatically.

**Using STL Data Structures to Avoid Memory Allocation:** A remarkably large amount of memory allocation and deallocation arises when dealing with two very common dynamic structures, vectors and lists. By *vector*, I mean a variable-sized array (which may be appended to), and by *list*, I mean a doubly linked list. Rather than going through the hassles of defining your own vector and list structures, and dealing with the headaches of memory allocation and deallocation, it is much simpler to use the built-in vector and list types provided by the C++ Standard Template Library, or STL. The STL provides data structures for a number standard containers, such as stacks, queues, deques (double ended queues), vectors, lists, priority queues, and maps.

One of the important features of the STL is that each can store objects of any one type. Such a class whose definition depends on a user-specified type is called a *template*. The type of the object being stored in the container is given in angle brackets. For example, we could define vectors to hold 100 integers or 500 characters as follows:

```cpp
#include <vector> // class vector definitions
using namespace std; // make std accessible
vector<int> scores(100); // a vector of (initially) 100 integers
vector<char> buffer(500); // a vector of (initially) 500 characters
```

Here are a couple of examples of how to use an STL vector.

```cpp
int n = 100;
vector<int> myInts(n); // allocate a vector with n ints
vector<Vector2d> myVects(n); // allocate a vector with n Vector2d objects
myInts[5] = 14; // you can use "[]" to index entries
myVects[7] = Vector2d(3,-4);
myVects[7].setX(2);
myVects.push_back(Vector2d(1,5)); // you can append entries to the vector
```

STL vectors and lists provide too many capabilities to be listed here. I will refer you to online documentation for more details. There are a number of other useful STL data structures, including dictionaries and priority queues.

STL vectors are superior to standard C++ arrays in many respects. First, as with arrays, individual elements can be indexed using the usual index operator ([ ]). They can also be accessed by the `at` member function, which also performs array bounds checking. (As in C, arrays in C++ do not even know their size, and hence range checking is not even possible.) In contrast, a vector object's size is given by its `size` member function. Unlike standard arrays, one vector object can be assigned to another, which results in the contents of one vector object being copied to the other. A vector can be resized dynamically by calling the `resize` member function. Here are some examples:

```cpp
int i = // ...;
    cout << scores[i]; // index (range unchecked)
buffer.at(i) = buffer.at(2 * i); // index (range checked)
vector<int> newScores = scores; // copy scores to newScores
scores.resize(scores.size() + 10); // add room for 10 more elements
```

Another handy STL container is the list, which implements a doubly linked list. Here is how to declare a list of floats. By default, the initial list is empty.

```cpp
```
List supports operations such as size (number of elements), empty (is the list empty?), front/back (return reference to first/last elements), push_front/push_back (insert to front/back), pop_front/pop_back (remove from front/back).

Iterators: The STL container classes introduced above all define a special associated class called an iterator. An iterator is an object that specifies a position within a container and which is endowed with the ability to navigate to other positions. If \( p \) is an iterator that refers to some position within a container, then \( *p \) yields a reference to the associated element.

Advancing to the next element of the container is done by incrementing the iterator. For example, either \( ++p \) or \( p++ \) advances \( p \) to point to the next element of the container. The former returns the updated value of the iterator, and the latter returns its original value. Each STL container class provides two member functions, begin and end, each of which returns an iterator for this container. The first returns an iterator that points to the first element of the container, and the second returns an iterator that can be thought of as pointing to an imaginary element just beyond the last element of the container.

Let us see how we can use iterators to enumerate the elements of an STL container \( C \). Suppose, for example, that \( C \) is of type \( \text{vector<int>} \), that is, it is an STL list of integers. The associated iterator type is denoted \( \text{vector<int>::iterator} \). For example, the code below demonstrates how to sum the elements of an STL vector \( V \) using an iterator.

```cpp
vector<int> v;
typedef vector<int>::iterator Iterator;  // iterator type
int sum = 0;
for (Iterator p = v.begin(); p != v.end(); ++p) {
    sum += *p;
}
cout << "The sum of elements in the list is " << sum << "\n";
```

Different containers provide iterators with different capabilities. Most STL containers (including lists, sets, and maps) provide the ability to move not only forward, but backward as well. For such containers the decrement operators \(--p \) and \( p-- \) are also defined for their iterators. This is called a bidirectional iterator.

A few STL containers (including vectors and deques) support the additional feature of allowing the addition and subtraction of an integer. For example, for such an iterator, \( p \), the value \( p + 3 \) references the element three positions after \( p \) in the container. This is called a random-access iterator. As with pointers, care is needed in the use of iterators. For example, it is up to the programmer to be sure that an iterator points to a valid element of the container before attempting to dereference it. Attempting to dereference an invalid iterator can result in your program aborting. As mentioned earlier, iterators can be invalid for various reasons. For example, an iterator becomes invalid if the position that it refers to is deleted.

References: In C, all parameter passing to functions is performed by value. This has two important implications. First, altering the value of an formal parameter inside a function has no effect on the actual parameter in the calling function. If you want to modify the value of a parameter, you need to pass a pointer to the parameter. This is messy, since it implies that the function needs to dereference the resulting parameter whenever it uses it. Second, passing a large class or structure to a function means that its entire contents will be copied. This can be inefficient for very large structures. (Note that this does not apply to C++ arrays, however, since an array is just a pointer to its first element. However, this would apply if you were to pass an STL vector by value. Such an operation would involve making a duplicate copy of the entire vector.)
In Java, this was handled very elegantly by making all object variables references. Thus, small primitive types, such as int and float are passed by value, and all objects are passed by reference. If it is desired to change the value of a primitive type, standard wrappers, like Integer, were defined.

In C++, this issue was addressed by defining a special type, called a reference. The following line defines the variable \( i \) to be an integer, and \( r \) to be a reference to this integer. All references to \( r \) are effectively “aliases” to references to \( i \).

```cpp
int i = 34;
int& r = i; // r is an alias for i
r = 27; // this is equivalent to i = 27
```

References are rarely used as shown above. Instead, they are principally used for passing parameters, “by reference” to functions. A reference parameter has two advantages over a value parameter. First, it can be modified (without any messy pointer dereferencing) and it is more efficient for class objects, since only the address of the object (not the entire object) needs to be conveyed to the function.

```cpp
void f(int& r, Vector2d& u) {
    r = 27; // changes the actual parameter to 27
    u.setY(4); // alters y-coordinate of actual parameter
}
```

// ... in your main program
int i = 34;
Vector2d v(2, 1);
f(i, v);
c out << i << " " << v.getY() << "\n"; // outputs "27 4"

Although passing class objects by reference is more efficient than by value, it has the downside that the function may inadvertently modify the value of the parameter, without the compiler being able to detect it. To handle this, C++ allows for something called a constant reference, which is a reference to an object that can be read, but not modified. Since the above function does not modify its arguments, it would more aptly be written in the following form:

```cpp
Vector2d add(const Vector2d& u, const Vector2d& v) {
    // ... in this function u and v may be read, but not modified
}
```

The Latest: The C++ standard continues to be upgraded from year to year. The latest version of the C++ standard is called C++11. It has a number of nice features. For example, the messy iterator syntax given above for enumerating the elements of a vector can be expressed much more succinctly as follows.

```cpp
vector<int> v;
// ...
for (int x : v) sum += x;
```

Unfortunately, as of now, only the very latest C++ compilers have this feature.

More information: There are a number of resources on the web, which can be found by searching for “C++ for Java programmers”. Here are some examples:

http://www.cs.williams.edu/~lenhart/courses/cs371/c++forjava.html
http://www.icce.rug.nl/documents/cplusplus/