**Jonathan Ruiz**

* **How many hours a week would you be available to commit to?**
  + Roughly 8 hours, however, I’m willing to work weekends if possible
  + During the summer, I’m willing to work 35 hours a week
* **Do you require visa sponsorship?**
  + I do not require a visa sponsorship

**1. Can you describe in detail and your own words how to accomplish runtime polymorphism in C++ with code examples?**

In C++, Polymorphism occurs by inheritance when a class hierarchy exists. An object that is Polymorphic can use the methods of the class that it is inheriting its methods and variables from. Runtime Polymorphism occurs through a class member method being overridden and this is achieved using pointers and virtual functions. If we create a pointer to an object that has a virtual function in it, that function will be bound at runtime and if we make the pointer point to an object that is derived from the base class with the overridden virtual function, it will execute the derived object’s function.

If the class method is not declared as virtual, then the derived class will use the method of the base class instead.

|  |
| --- |
| #include <iostream>  class Mage {  public:  void spell() {  std::cout << "Fireball!!";  }  };  class Necromancer : public Mage{  public:  void spell() {  std::cout << "Rise from the dead";  }  };  int main() {  Mage\* p1 = new Mage();  p1 = new Necromancer;  p1->spell();  } |

Like so:

|  |
| --- |
|  |

And if you declare the method as virtual

|  |
| --- |
| #include <iostream>  class Mage {  public:  void virtual spell() {  std::cout << "Fireball!!";  }  };  class Necromancer : public Mage{  public:  void spell() {  std::cout << "Rise from the dead";  }  };  int main() {  Mage\* p1 = new Mage();  p1 = new Necromancer;  p1->spell();  } |

You get the following output:

|  |
| --- |
|  |

**2.           Can you read the following code and explain to us the purpose of foo?  Are there bugs you’d fix or ways you would suggest to improve foo?  Please send us your improved/corrected version.**

The function foo takes a C++ vector of 3D points and goes through pointi and point j (which is the point right next topoint i in the vector). It then assigns tmp1 and tmp2, the magnitude of point i and j respectively and checks if the magnitude of tmp2 (the point after tmp1) is less than tmp1. If it is, it swaps the points in the vector and sets done to false so that the for loop keeps running. In short, foo simply attempts to sort the vector of 3D points by magnitude, using an optimized bubble sort.

The first problem occurs when i = points.size() – 1 because j gets assigned i+1 which is points.size(). This will give an out of range or index out of bounds error. A simple solution to this would be to just change the for loop condition to check i < points.size() – 1, so that i can never be points.size() – 1 and j can never be points.size(). And to improve readability, you could set a variable to be points.size() – 1 and use that variable in the loop.

The next problem is that the function does not actually change the points vector or return the result of the sorted vector. I think there are two solutions to this depending on what exactly needs to be done. The very simple and more efficient solution would be to just change the pass of the points vector, to a pass by reference. The other solution would be to change the return type of foo to a vector<ThreeDPoint> and assign the result of foo to the original points vector. This is less efficient, however, since we’re creating a new vector of type ThreeDPoint just to return the result of the changed points vector. It’s easier and more efficient to pass by reference. The final change that could be considered would be to change the sorting algorithm since bubble sort is inefficient with large amounts of data but assuming the size of vectors is kept to a small scale, bubble sort is perfectly acceptable.

There are a few minor tweaks that I also made to the source code, such as adding a magnitude function just to slightly improve readability and not have to type out the entire magnitude formula every time in case it needed to be used again. The definition and declaration of j was moved into the initialization and the increment was moved into the incrementor of the for loop since it only increases by 1 every iteration in the for loop. Below is the modified code without the main:

|  |
| --- |
| #include <vector>  #include <iostream>  #include <chrono>  struct ThreeDPoint  {  int x;  int y;  int z;  };  // magnitude function used for extra readability  float magnitude(ThreeDPoint point) { // Finding length of vector  return (float)sqrt(point.x \* point.x + point.y \* point.y + point.z \* point.z);  }  void foo(std::vector<ThreeDPoint>& points)  {  float tmp1 = 0.0f;  float tmp2 = 0.0f;  bool done = false;  size\_t p\_size = points.size() - 1;  while (!done)  {  done = true;  for (int i = 0, j = i + 1; i < p\_size; ++i, ++j) {  tmp1 = magnitude(points[i]);  tmp2 = magnitude(points[j]);  if (tmp2 < tmp1) {  std::swap(points[i], points[j]);  done = false;  }  }  }  } |

If the size of the points vector becomes too large then it’s worth considering using a merge sort or a quicksort. I’ve included a .cpp file that has this optimized bubble sort, a merge sort, and a quick sort implementation. Along with the sorts, I’ve included tests that time each sort on the same vector of 3D points.

**3.           Can you implement IsValidPlayer in the example below for us in C++?**

Assume the size and contents of the array won't change during execution but do not assume they are the example data/size provided below.  IsValidPlayer should return true if the passed in string matches any string in the PlayerNames array, otherwise return false.

If the data is constant during execution and is not sorted, the easiest solution would be to perform a simple linear search like below. The time taken to look a player name up is not affected greatly by this and to further show this, I performed a rudimentary test. I ran 10,000,000 iterations of 14 lookups through the array in 10 iterations. In addition to this, I added more PlayerNames to the array and a few of the lookups would always look through the whole array. The number of milliseconds taken per 140,000,000 lookups ended up being around ~2.6 seconds, or 2600 milliseconds. I’ve attached the .cpp file that I used to test which used the chrono library for the timings. If the data was sorted before code execution, however, a binary search would be appropriate.

|  |
| --- |
| #include <iostream>  **const** **char**\* PlayerNames[] =  {                 "Scorpion",                 "SubZero",                 "LiuKang"  };  bool IsValidPlayer(const char\* player)  {  for (const char\* p : PlayerNames) {  if (p == player) return true;  }  return false;  } |