12

C and C++

A good interviewer won't demand that you code in a language you don't profess to know. Hopefully, if you're asked to code in C++, it's listed on your resume. If you don't remember all the APIs, don't worry—most interviewers (though not all) don't care that much. We do recommend, however, studying up on basic C++ syntax so that you can approach these questions with ease.

▸ Classes and Inheritance

Though C++ classes have similar characteristics to those of other languages, we'll review some of the syntax below.

The code below demonstrates the implementation of a basic class with inheritance.

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

#define NAME_SIZE 50 // Defines a macro

class Person {
  int id; // all members are private by default
  char name[NAME_SIZE];

 public:
  void aboutMe() {
     cout << "I am a person.";
  }
};

class Student : public Person {
 public:
  void aboutMe() {
     cout << "I am a student.";
  }
};

int main() {
  Student * p = new Student();
  p->aboutMe(); // prints "I am a student."
  delete p; // Important! Make sure to delete allocated memory.
  return 0;
}
```

All data members and methods are private by default in C++. One can modify this by introducing the keyword `public`.

▸ Constructors and Destructors

The constructor of a class is automatically called upon an object's creation. If no constructor is defined, the compiler automatically generates one called the Default Constructor. Alternatively, we can define our own constructor.

If you just need to initialize primitive types, a simple way to do it is this:

```
Person(int a) {
  id = a;
}
```

This works for primitive types, but you might instead want to do this:

```
Person(int a) : id(a) {
  ...
}
```

The data member `id` is assigned before the actual object is created and before the remainder of the constructor code is called. This approach is necessary when the fields are constant or class types.

The destructor cleans up upon object deletion and is automatically called when an object is destroyed. It cannot take an argument as we don't explicitly call a destructor.

```
~Person() {
  delete obj; // free any memory allocated within class
}
```

▸ Virtual Functions

In an earlier example, we defined p to be of type `Student`:

```
Student * p = new Student();
p->aboutMe();
```

What would happen if we defined p to be a `Person*`, like so?

```
Person * p = new Student();
p->aboutMe();
```

In this case, "`I am a person`" would be printed instead. This is because the function `aboutMe` is resolved at compile-time, in a mechanism known as *static binding*.

If we want to ensure that the `Student`'s implementation of `aboutMe` is called, we can define `aboutMe` in the `Person` class to be `virtual`.

```
class Person {
  ...
  virtual void aboutMe() {
    cout << "I am a person.";
  }
};

class Student : public Person {
 public:
  void aboutMe() {
    cout << "I am a student.";
  }
};
```

Another usage for virtual functions is when we can't (or don't want to) implement a method for the parent class. Imagine, for example, that we want Student and Teacher to inherit from Person so that we can implement a common method such as addCourse(string s). Calling addCourse on Person, however, wouldn't make much sense since the implementation depends on whether the object is actually a Student or Teacher.

In this case, we might want addCourse to be a virtual function defined within Person, with the implementation being left to the subclass.

```
class Person {
   int id; // all members are private by default
   char name[NAME_SIZE];
  public:
   virtual void aboutMe() {
      cout << "I am a person." << endl;
   }
   virtual bool addCourse(string s) = 0;
};

class Student : public Person {
  public:
   void aboutMe() {
      cout << "I am a student." << endl;
   }

   bool addCourse(string s) {
      cout << "Added course " << s << " to student." << endl;
      return true;
   }
};

int main() {
   Person * p = new Student();
   p->aboutMe(); // prints "I am a student."
   p->addCourse("History");
   delete p;
}
```

Note that by defining addCourse to be a "pure virtual function," Person is now an abstract class and we cannot instantiate it.

▸ Virtual Destructor

The virtual function naturally introduces the concept of a "virtual destructor." Suppose we wanted to implement a destructor method for Person and Student. A naive solution might look like this:

```
class Person {
 public:
   ~Person() {
      cout << "Deleting a person." << endl;
   }
};

class Student : public Person {
 public:
  ~Student() {
     cout << "Deleting a student." << endl;
  }
};

int main() {
  Person * p = new Student();
  delete p; // prints "Deleting a person."
}
```

As in the earlier example, since p is a Person, the destructor for the Person class is called. This is problematic because the memory for Student may not be cleaned up.

To fix this, we simply define the destructor for Person to be virtual.

```
class Person {
 public:
  virtual ~Person() {
     cout << "Deleting a person." << endl;
  }
};

class Student : public Person {
 public:
  ~Student() {
     cout << "Deleting a student." << endl;
  }
};

int main() {
  Person * p = new Student();
  delete p;
}
```

This will output the following:

```
Deleting a student.
Deleting a person.
```

▸ Default Values

Functions can specify default values, as shown below. Note that all default parameters must be on the right side of the function declaration, as there would be no other way to specify how the parameters line up.

```
int func(int a, int b = 3) {
  x = a;
  y = b;
  return a + b;
}

w = func(4);
z = func(4, 5);
```

▸ Operator Overloading

Operator overloading enables us to apply operators like + to objects that would otherwise not support these operations. For example, if we wanted to merge two BookShelves into one, we could overload the + operator as follows.

```
BookShelf BookShelf::operator+(BookShelf &other) { ... }
```

▸ Pointers and References

A pointer holds the address of a variable and can be used to perform any operation that could be directly done on the variable, such as accessing and modifying it.

Two pointers can equal each other, such that changing one's value also changes the other's value (since they, in fact, point to the same address).

```
int * p = new int;
*p = 7;
int * q = p;
*p = 8;
cout << *q; // prints 8
```

Note that the size of a pointer varies depending on the architecture: 32 bits on a 32-bit machine and 64 bits on a 64-bit machine. Pay attention to this difference, as it's common for interviewers to ask exactly how much space a data structure takes up.

References

A reference is another name (an alias) for a pre-existing object and it does not have memory of its own. For example:

```
1  int a = 5;
2  int & b = a;
3  b = 7;
4  cout << a; // prints 7
```

In line 2 above, b is a reference to a; modifying b will also modify a.

You cannot create a reference without specifying where in memory it refers to. However, you can create a free-standing reference as shown below:

```
/* allocates memory to store 12 and makes b a reference to this
 * piece of memory. */
const int & b = 12;
```

Unlike pointers, references cannot be null and cannot be reassigned to another piece of memory.

Pointer Arithmetic

One will often see programmers perform addition on a pointer, such as what you see below:

```
int * p = new int[2];
p[0] = 0;
p[1] = 1;
p++;
cout << *p; // Outputs 1
```

Performing p++ will skip ahead by `sizeof(int)` bytes, such that the code outputs 1. Had p been of different type, it would skip ahead as many bytes as the size of the data structure.

▸ Templates

Templates are a way of reusing code to apply the same class to different data types. For example, we might have a list-like data structure which we would like to use for lists of various types. The code below implements this with the `ShiftedList` class.

```
template <class T>class ShiftedList {
  T* array;
  int offset, size;
 public:
  ShiftedList(int sz) : offset(0), size(sz) {
    array = new T[size];
  }

  ~ShiftedList() {
    delete [] array;
  }

  void shiftBy(int n) {
    offset = (offset + n) % size;
  }

  T getAt(int i) {
    return array[convertIndex(i)];
  }

  void setAt(T item, int i) {
    array[convertIndex(i)] = item;
  }

private:
  int convertIndex(int i) {
    int index = (i - offset) % size;
    while (index < 0) index += size;
    return index;
  }
};
```

Interview Questions

12.1 **Last K Lines:** Write a method to print the last K lines of an input file using C++.

Hints: #449, #459

pg 422

12.2 **Reverse String:** Implement a function `void reverse(char* str)` in C or C++ which reverses a null-terminated string.

Hints: #410, #452

pg 423

12.3 **Hash Table vs. STL Map:** Compare and contrast a hash table and an STL map. How is a hash table implemented? If the number of inputs is small, which data structure options can be used instead of a hash table?

Hints: #423

pg 423

12.4 Virtual Functions: How do virtual functions work in C++?

Hints: #463

pg 424

12.5 Shallow vs. Deep Copy: What is the difference between deep copy and shallow copy? Explain how you would use each.

Hints: #445

pg 425

12.6 Volatile: What is the significance of the keyword "volatile" in C?

Hints: #456

pg 426

12.7 Virtual Base Class: Why does a destructor in base class need to be declared `virtual`?

Hints: #421, #460

pg 427

12.8 Copy Node: Write a method that takes a pointer to a `Node` structure as a parameter and returns a complete copy of the passed in data structure. The `Node` data structure contains two pointers to other `Nodes`.

Hints: #427, #462

pg 427

12.9 Smart Pointer: Write a smart pointer class. A smart pointer is a data type, usually implemented with templates, that simulates a pointer while also providing automatic garbage collection. It automatically counts the number of references to a `SmartPointer<T*>` object and frees the object of type T when the reference count hits zero.

Hints: #402, #438, #453

pg 428

12.10 Malloc: Write an aligned malloc and free function that supports allocating memory such that the memory address returned is divisible by a specific power of two.

EXAMPLE

`align_malloc(1000,128)` will return a memory address that is a multiple of 128 and that points to memory of size 1000 bytes.

`aligned_free()` will free memory allocated by `align_malloc`.

Hints: #413, #432, #440

pg 430

12.11 2D Alloc: Write a function in C called `my2DAlloc` which allocates a two-dimensional array. Minimize the number of calls to `malloc` and make sure that the memory is accessible by the notation `arr[i][j]`.

Hints: #406, #418, #426

pg 431

Additional Questions: Linked Lists (#2.6), Testing (#11.1), Java (#13.4), Threads and Locks (#15.3).

Hints start on page 676.