```
13 };
```

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.

```
1  class Person {
2    int id; // all members are private by default
3    char name[NAME_SIZE];
4    public:
5    virtual void aboutMe() {
6      cout << "I am a person." << endl;
7    }
8    virtual bool addCourse(string s) = 0;
9  };
10
11 class Student : public Person {
12   public:
13   void aboutMe() {
14     cout << "I am a student." << endl;
15   }
16
17   bool addCourse(string s) {
18     cout << "Added course " << s << " to student." << endl;
19     return true;
20   }
21 };
22
23 int main() {
24   Person * p = new Student();
25   p->aboutMe(); // prints "I am a student."
26   p->addCourse("History");
27   delete p;
28 }
```

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:

```
1  class Person {
2   public:
3    ~Person() {
4      cout << "Deleting a person." << endl;
5    }
6  };
7
8  class Student : public Person {
9   public:
```

```
10    ~Student() {
11       cout << "Deleting a student." << endl;
12    }
13 };
14
15 int main() {
16    Person * p = new Student();
17    delete p; // prints "Deleting a person."
18 }
```

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.

```
1  class Person {
2   public:
3    virtual ~Person() {
4       cout << "Deleting a person." << endl;
5    }
6  };
7
8  class Student : public Person {
9   public:
10   ~Student() {
11      cout << "Deleting a student." << endl;
12   }
13 };
14
15 int main() {
16   Person * p = new Student();
17   delete p;
18 }
```

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.

```
1  int func(int a, int b = 3) {
2    x = a;
3    y = b;
4    return a + b;
5  }
6
7  w = func(4);
8  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.

```
1  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).

```
1  int * p = new int;
2  *p = 7;
3  int * q = p;
4  *p = 8;
5  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:

```
1  /* allocates memory to store 12 and makes b a reference to this
2   * piece of memory. */
3  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:

```
1  int * p = new int[2];
2  p[0] = 0;
3  p[1] = 1;
4  p++;
5  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.

13

Java

While Java-related questions are found throughout this book, this chapter deals with questions about the language and syntax. Such questions are more unusual at bigger companies, which believe more in testing a candidate's aptitude than a candidate's knowledge (and which have the time and resources to train a candidate in a particular language). However, at other companies, these pesky questions can be quite common.

How to Approach

As these questions focus so much on knowledge, it may seem silly to talk about an approach to these problems. After all, isn't it just about knowing the right answer?

Yes and no. Of course, the best thing you can do to master these questions is to learn Java inside and out. But, if you do get stumped, you can try to tackle it with the following approach:

1. Create an example of the scenario, and ask yourself how things should play out.
2. Ask yourself how other languages would handle this scenario.
3. Consider how you would design this situation if you were the language designer. What would the implications of each choice be?

Your interviewer may be equally—or more—impressed if you can derive the answer than if you automatically knew it. Don't try to bluff though. Tell the interviewer, "I'm not sure I can recall the answer, but let me see if I can figure it out. Suppose we have this code..."

Overloading vs. Overriding

Overloading is a term used to describe when two methods have the same name but differ in the type or number of arguments.

```
1  public double computeArea(Circle c) { ... }
2  public double computeArea(Square s) { ... }
```

Overriding, however, occurs when a method shares the same name and function signature as another method in its super class.

```
1  public abstract class Shape {
2    public void printMe() {
3      System.out.println("I am a shape.");
4    }
5    public abstract double computeArea();
6  }
```

```
7
8  public class Circle extends Shape {
9     private double rad = 5;
10    public void printMe() {
11       System.out.println("I am a circle.");
12    }
13
14    public double computeArea() {
15       return rad * rad * 3.15;
16    }
17 }
18
19 public class Ambiguous extends Shape {
20    private double area = 10;
21    public double computeArea() {
22       return area;
23    }
24 }
25
26 public class IntroductionOverriding {
27    public static void main(String[] args) {
28       Shape[] shapes = new Shape[2];
29       Circle circle = new Circle();
30       Ambiguous ambiguous = new Ambiguous();
31
32       shapes[0] = circle;
33       shapes[1] = ambiguous;
34
35       for (Shape s : shapes) {
36          s.printMe();
37          System.out.println(s.computeArea());
38       }
39    }
40 }
```

The above code will print:

```
1  I am a circle.
2  78.75
3  I am a shape.
4  10.0
```

Observe that `Circle` overrode `printMe()`, whereas `Ambiguous` just left this method as-is.

▸ Collection Framework

Java's collection framework is incredibly useful, and you will see it used throughout this book. Here are some of the most useful items:

`ArrayList`: An `ArrayList` is a dynamically resizing array, which grows as you insert elements.

```
1  ArrayList<String> myArr = new ArrayList<String>();
2  myArr.add("one");
3  myArr.add("two");
4  System.out.println(myArr.get(0)); / *prints <one> */
```

`Vector`: A `vector` is very similar to an `ArrayList`, except that it is synchronized. Its syntax is almost identical as well.

```
Vector<String> myVect = new Vector<String>();
myVect.add("one");
myVect.add("two");
System.out.println(myVect.get(0));
```

`LinkedList`: `LinkedList` is, of course, Java's built-in `LinkedList` class. Though it rarely comes up in an interview, it's useful to study because it demonstrates some of the syntax for an iterator.

```
LinkedList<String> myLinkedList = new LinkedList<String>();
myLinkedList.add("two");
myLinkedList.addFirst("one");
Iterator<String> iter = myLinkedList.iterator();
while (iter.hasNext()) {
   System.out.println(iter.next());
}
```

HashMap: The `HashMap` collection is widely used, both in interviews and in the real world. We've provided a snippet of the syntax below.

```
HashMap<String, String> map = new HashMap<String, String>();
map.put("one", "uno");
map.put("two", "dos");
System.out.println(map.get("one"));
```

Before your interview, make sure you're very comfortable with the above syntax. You'll need it.

Interview Questions

Please note that because virtually all the solutions in this book are implemented with Java, we have selected only a small number of questions for this chapter. Moreover, most of these questions deal with the "trivia" of the languages, since the rest of the book is filled with Java programming questions.

13.1 **Private Constructor:** In terms of inheritance, what is the effect of keeping a constructor private?

Hints: #404

pg 433

13.2 **Return from Finally:** In Java, does the `finally` block get executed if we insert a `return` statement inside the try block of a `try-catch-finally`?

Hints: #409

pg 433

13.3 **Final, etc.:** What is the difference between `final`, `finally`, and `finalize`?

Hints: #412

pg 433

13.4 **Generics vs. Templates:** Explain the difference between templates in C++ and generics in Java.

Hints: #416, #425

pg 435

13.5 **TreeMap, HashMap, LinkedHashMap:** Explain the differences between `TreeMap`, `HashMap`, and `LinkedHashMap`. Provide an example of when each one would be best.

Hints: #420, #424, #430, #454

pg 436

13.6 **Object Reflection:** Explain what object reflection is in Java and why it is useful.

Hints: #435

pg 437

13.7 **Lambda Expressions:** There is a class `Country` that has methods `getContinent()` and `getPopulation()`. Write a function `int getPopulation(List<Country> countries, String continent)` that computes the total population of a given continent, given a list of all countries and the name of a continent.

Hints: #448, #461, #464

pg 438

13.8 **Lambda Random:** Using Lambda expressions, write a function `List<Integer> getRandomSubset(List<Integer> list)` that returns a random subset of arbitrary size. All subsets (including the empty set) should be equally likely to be chosen.

Hints: #443, #450, #457

pg 439

Additional Questions: Arrays and Strings (#1.3), Object-Oriented Design (#7.12), Threads and Locks (#15.3)

Hints start on page 676.

14

Databases

If you profess knowledge of databases, you might be asked some questions on it. We'll review some of the key concepts and offer an overview of how to approach these problems. As you read these queries, don't be surprised by minor variations in syntax. There are a variety of flavors of SQL, and you might have worked with a slightly different one. The examples in this book have been tested against Microsoft SQL Server.

SQL Syntax and Variations

Implicit and explicit joins are shown below. These two statements are equivalent, and it's a matter of personal preference which one you choose. For consistency, we will stick to the explicit join.

Explicit Join

```
1  SELECT CourseName, TeacherName
2  FROM Courses INNER JOIN Teachers
3  ON Courses.TeacherID = Teachers.TeacherID
```

Implicit Join

```
1  SELECT CourseName, TeacherName
2  FROM   Courses, Teachers
3  WHERE  Courses.TeacherID =
4         Teachers.TeacherID
```

Denormalized vs. Normalized Databases

Normalized databases are designed to minimize redundancy, while denormalized databases are designed to optimize read time.

In a traditional normalized database with data like `Courses` and `Teachers`, `Courses` might contain a column called `TeacherID`, which is a foreign key to `Teacher`. One benefit of this is that information about the teacher (name, address, etc.) is only stored once in the database. The drawback is that many common queries will require expensive joins.

Instead, we can denormalize the database by storing redundant data. For example, if we knew that we would have to repeat this query often, we might store the teacher's name in the `Courses` table. Denormalization is commonly used to create highly scalable systems.

SQL Statements

Let's walk through a review of basic SQL syntax, using as an example the database that was mentioned earlier. This database has the following simple structure (* indicates a primary key):

```
Courses: CourseID*, CourseName, TeacherID
Teachers: TeacherID*, TeacherName
Students: StudentID*, StudentName
```