11.5 Test a Pen: How would you test a pen?

pg 157

SOLUTION

This problem is largely about understanding the constraints and approaching the problem in a structured manner.

To understand the constraints, you should ask a lot of questions to understand the "who, what, where, when, how and why" of a problem (or as many of those as apply to the problem). Remember that a good tester understands exactly what he is testing before starting the work.

To illustrate the technique in this problem, let us guide you through a mock conversation.

- **Interviewer:** How would you test a pen?
- **Candidate:** Let me find out a bit about the pen. Who is going to use the pen?
- **Interviewer:** Probably children.
- **Candidate:** Okay, that's interesting. What will they be doing with it? Will they be writing, drawing, or doing something else with it?
- **Interviewer:** Drawing.
- **Candidate:** Okay, great. On what? Paper? Clothing? Walls?
- **Interviewer:** On clothing.
- **Candidate:** Great. What kind of tip does the pen have? Felt? Ballpoint? Is it intended to wash off, or is it intended to be permanent?
- **Interviewer:** It's intended to wash off.

Many questions later, you may get to this:

- **Candidate:** Okay, so as I understand it, we have a pen that is being targeted at 5 to 10-year-olds. The pen has a felt tip and comes in red, green, blue and black. It's intended to wash off when clothing is washed. Is that correct?

The candidate now has a problem that is significantly different from what it initially seemed to be. This is not uncommon. In fact, many interviewers intentionally give a problem that seems clear (everyone knows what a pen is!), only to let you discover that it's quite a different problem from what it seemed. Their belief is that users do the same thing, though users do so accidentally.

Now that you understand what you're testing, it's time to come up with a plan of attack. The key here is *structure*.

Consider what the different components of the object or problem, and go from there. In this case, the components might be:

- *Fact check:* Verify that the pen is felt tip and that the ink is one of the allowed colors.
- *Intended use:* Drawing. Does the pen write properly on clothing?
- *Intended use:* Washing. Does it wash off of clothing (even if it's been there for an extended period of time)? Does it wash off in hot, warm and cold water?
- *Safety:* Is the pen safe (non-toxic) for children?
- *Unintended uses:* How else might children use the pen? They might write on other surfaces, so you need to check whether the behavior there is correct. They might also stomp on the pen, throw it, and so on.

You'll need to make sure that the pen holds up under these conditions.

Remember that in any testing question, you need to test both the intended and unintended scenarios. People don't always use the product the way you want them to.

11.6 Test an ATM: How would you test an ATM in a distributed banking system?

pg 157

SOLUTION

The first thing to do on this question is to clarify assumptions. Ask the following questions:

- Who is going to use the ATM? Answers might be "anyone," or it might be "blind people," or any number of other answers.
- What are they going to use it for? Answers might be "withdrawing money," "transferring money," "checking their balance," or many other answers.
- What tools do we have to test? Do we have access to the code, or just to the ATM?

Remember: a good tester makes sure she knows what she's testing!

Once we understand what the system looks like, we'll want to break down the problem into different testable components. These components include:

- Logging in
- Withdrawing money
- Depositing money
- Checking balance
- Transferring money

We would probably want to use a mix of manual and automated testing.

Manual testing would involve going through the steps above, making sure to check for all the error cases (low balance, new account, nonexistent account, and so on).

Automated testing is a bit more complex. We'll want to automate all the standard scenarios, as shown above, and we also want to look for some very specific issues, such as race conditions. Ideally, we would be able to set up a closed system with fake accounts and ensure that, even if someone withdraws and deposits money rapidly from different locations, he never gets money or loses money that he shouldn't.

Above all, we need to prioritize security and reliability. People's accounts must always be protected, and we must make sure that money is always properly accounted for. No one wants to unexpectedly lose money! A good tester understands the system priorities.

12

Solutions to C and C++

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

pg 163

SOLUTION

One brute force way could be to count the number of lines (N) and then print from N-K to Nth line. But this requires two reads of the file, which is unnecessarily costly. We need a solution which allows us to read just once and be able to print the last K lines.

We can allocate an array for all K lines and the last K lines we've read in the array. , and so on. Each time that we read a new line, we purge the oldest line from the array.

But—you might ask—wouldn't this require shifting elements in the array, which is also very expensive? No, not if we do it correctly. Instead of shifting the array each time, we will use a circular array.

With a circular array, we always replace the oldest item when we read a new line. The oldest item is tracked in a separate variable, which adjusts as we add new items.

The following is an example of a circular array:

```
step 1 (initially): array = {a, b, c, d, e, f}. p = 0
step 2 (insert g):  array = {g, b, c, d, e, f}. p = 1
step 3 (insert h):  array = {g, h, c, d, e, f}. p = 2
step 4 (insert i):  array = {g, h, i, d, e, f}. p = 3
```

The code below implements this algorithm.

```
1  void printLast10Lines(char* fileName) {
2    const int K = 10;
3    ifstream file (fileName);
4    string L[K];
5    int size = 0;
6
7    /* read file line by line into circular array */
8    /* peek() so an EOF following a line ending is not considered a separate line */
9    while (file.peek() != EOF) {
10     getline(file, L[size % K]);
11     size++;
12   }
13
14   /* compute start of circular array, and the size of it */
15   int start = size > K ? (size % K) : 0;
16   int count = min(K, size);
17
```

```
    /* print elements in the order they were read */
    for (int i = 0; i < count; i++) {
       cout << L[(start + i) % K] << endl;
    }
}
```

This solution will require reading in the whole file, but only ten lines will be in memory at any given point.

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

pg 163

SOLUTION

This is a classic interview question. The only "gotcha" is to try to do it in place, and to be careful for the `null` character.

We will implement this in C.

```
void reverse(char *str) {
   char* end = str;
   char tmp;
   if (str) {
      while (*end) { /* find end of the string */
         ++end;
      }
      --end; /* set one char back, since last char is null */

      /* swap characters from start of string with the end of the string, until the
       * pointers meet in middle. */
      while (str < end) {
         tmp = *str;
         *str++ = *end;
         *end-- = tmp;
      }
   }
}
```

This is just one of many ways to implement this solution. We could even implement this code recursively (but we wouldn't recommend it).

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?

pg 163

SOLUTION

In a hash table, a value is stored by calling a hash function on a key. Values are not stored in sorted order. Additionally, since hash tables use the key to find the index that will store the value, an insert or lookup can be done in amortized `O(1)` time (assuming few collisions in the hash table). In a hash table, one must also handle potential collisions. This is often done by chaining, which means to create a linked list of all the values whose keys map to a particular index.

An STL map inserts the key/value pairs into a binary search tree based on the keys. There is no need to handle collisions, and, since the tree is balanced, the insert and lookup time is guaranteed to be `O(log N)`.

How is a hash table implemented?

A hash table is traditionally implemented with an array of linked lists. When we want to insert a key/value pair, we map the key to an index in the array using a hash function. The value is then inserted into the linked list at that position.

Note that the elements in a linked list at a particular index of the array do not have the same key. Rather, `hashFunction(key)` is the same for these values. Therefore, in order to retrieve the value for a specific key, we need to store in each node both the exact key and the value.

To summarize, the hash table will be implemented with an array of linked lists, where each node in the linked list holds two pieces of data: the value and the original key. In addition, we will want to note the following design criteria:

1. We want to use a good hash function to ensure that the keys are well distributed. If they are not well distributed, then we would get a lot of collisions and the speed to find an element would decline.
2. No matter how good our hash function is, we will still have collisions, so we need a method for handling them. This often means chaining via a linked list, but it's not the only way.
3. We may also wish to implement methods to dynamically increase or decrease the hash table size depending on capacity. For example, when the ratio of the number of elements to the table size exceeds a certain threshold, we may wish to increase the hash table size. This would mean creating a new hash table and transferring the entries from the old table to the new table. Because this is an expensive operation, we want to be careful to not do it too often.

What can be used instead of a hash table, if the number of inputs is small?

You can use an STL map or a binary tree. Although this takes `O(log(n))` time, the number of inputs may be small enough to make this time negligible.

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

pg 164

SOLUTION

A virtual function depends on a "vtable" or "Virtual Table." If any function of a class is declared to be virtual, a vtable is constructed which stores addresses of the virtual functions of this class. The compiler also adds a hidden `vptr` variable in all such classes which points to the vtable of that class. If a virtual function is not overridden in the derived class, the vtable of the derived class stores the address of the function in its parent class. The vtable is used to resolve the address of the function when the virtual function is called. Dynamic binding in C++ is performed through the vtable mechanism.

Thus, when we assign the derived class object to the base class pointer, the `vptr` variable points to the vtable of the derived class. This assignment ensures that the most derived virtual function gets called.

Consider the following code.

```
1  class Shape {
2    public:
3     int edge_length;
4     virtual int circumference () {
```

```
5       cout << "Circumference of Base Class\n";
6       return 0;
7     }
8   };
9
10  class Triangle: public Shape {
11    public:
12    int circumference () {
13       cout<< "Circumference of Triangle Class\n";
14       return 3 * edge_length;
15    }
16  };
17
18  void main() {
19    Shape * x = new Shape();
20    x->circumference(); // "Circumference of Base Class"
21    Shape *y = new Triangle();
22    y->circumference(); // "Circumference of Triangle Class"
23  }
```

In the previous example, `circumference` is a virtual function in the Shape class, so it becomes virtual in each of the derived classes (`Triangle`, etc). C++ non-virtual function calls are resolved at compile time with static binding, while virtual function calls are resolved at runtime with dynamic binding.

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

pg 164

SOLUTION

A shallow copy copies all the member values from one object to another. A deep copy does all this and also deep copies any pointer objects.

An example of shallow and deep copy is below.

```
1   struct Test {
2     char * ptr;
3   };
4
5   void shallow_copy(Test & src, Test & dest) {
6     dest.ptr = src.ptr;
7   }
8
9   void deep_copy(Test & src, Test & dest) {
10    dest.ptr = (char*)malloc(strlen(src.ptr) + 1);
11    strcpy(dest.ptr, src.ptr);
12  }
```

Note that `shallow_copy` may cause a lot of programming runtime errors, especially with the creation and deletion of objects. Shallow copy should be used very carefully and only when a programmer really understands what he wants to do. In most cases, shallow copy is used when there is a need to pass information about a complex structure without actual duplication of data. One must also be careful with destruction of objects in a shallow copy.

In real life, shallow copy is rarely used. Deep copy should be used in most cases, especially when the size of the copied structure is small.

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

pg 164

SOLUTION

The keyword `volatile` informs the compiler that the value of variable it is applied to can change from the outside, without any update done by the code. This may be done by the operating system, the hardware, or another thread. Because the value can change unexpectedly, the compiler will therefore reload the value each time from memory.

A volatile integer can be declared by either of the following statements:

```
int volatile x;
volatile int x;
```

To declare a pointer to a volatile integer, we do the following:

```
volatile int * x;
int volatile * x;
```

A volatile pointer to non-volatile data is rare, but can be done.

```
int * volatile x;
```

If you wanted to declare a volatile variable pointer for volatile memory (both pointer address and memory contained are volatile), you would do the following:

```
int volatile * volatile x;
```

Volatile variables are not optimized, which can be very useful. Imagine this function:

```
1  int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

At first glance, our code appears to loop infinitely. The compiler may try to optimize it to:

```
1  void Fn(void) {
2    start:
3      int opt = 1;
4      if (true)
5      goto start;
6  }
```

This becomes an infinite loop. However, an external operation might write '0' to the location of variable `opt`, thus breaking the loop.

To prevent the compiler from performing such optimization, we want to signal that another element of the system could change the variable. We do this using the `volatile` keyword, as shown below.

```
1  volatile int opt = 1;
2  void Fn(void) {
3    start:
4      if (opt == 1) goto start;
5      else break;
6  }
```

Volatile variables are also useful when multi-threaded programs have global variables and any thread can modify these shared variables. We may not want optimization on these variables.

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

pg 164

SOLUTION

Let's think about why we have virtual methods to start with. Suppose we have the following code:

```
1  class Foo {
2   public:
3     void f();
4  };
5
6  class Bar : public Foo {
7   public:
8     void f();
9  }
10
11 Foo * p = new Bar();
12 p->f();
```

Calling `p->f()` will result in a call to `Foo::f()`. This is because `p` is a pointer to `Foo`, and `f()` is not virtual.

To ensure that `p->f()` will invoke the most derived implementation of `f()`, we need to declare `f()` to be a virtual function.

Now, let's go back to our destructor. Destructors are used to clean up memory and resources. If Foo's destructor were not virtual, then Foo's destructor would be called, even when `p` is *really* of type `Bar`.

This is why we declare destructors to be virtual; we want to ensure that the destructor for the most derived class is called.

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 `Node`s.

pg 164

SOLUTION

The algorithm will maintain a mapping from a node address in the original structure to the corresponding node in the new structure. This mapping will allow us to discover previously copied nodes during a traditional depth-first traversal of the structure. Traversals often mark visited nodes—the mark can take many forms and does not necessarily need to be stored in the node.

Thus, we have a simple recursive algorithm:

```
1  typedef map<Node*, Node*> NodeMap;
2
3  Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
4    if (cur == NULL) {
5      return NULL;
6    }
7
8    NodeMap::iterator i = nodeMap.find(cur);
9    if (i != nodeMap.end()) {
10     // we've been here before, return the copy
11     return i->second;
12   }
```

```
13
14    Node * node = new Node;
15    nodeMap[cur] = node; // map current before traversing links
16    node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
17    node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
18    return node;
19  }
20
21  Node * copy_structure(Node * root) {
22    NodeMap nodeMap; // we will need an empty map
23    return copy_recursive(root, nodeMap);
24  }
```

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.

pg 164

SOLUTION

A smart pointer is the same as a normal pointer, but it provides safety via automatic memory management. It avoids issues like dangling pointers, memory leaks and allocation failures. The smart pointer must maintain a single reference count for all references to a given object.

This is one of those problems that seems at first glance pretty overwhelming, especially if you're not a C++ expert. One useful way to approach the problem is to divide the problem into two parts: (1) outline the pseudocode and approach and then (2) implement the detailed code.

In terms of the approach, we need a reference count variable that is incremented when we add a new reference to the object and decremented when we remove a reference. The code should look something like the below pseudocode:

```
1   template <class T> class SmartPointer {
2     /* The smart pointer class needs pointers to both the object itself and to the
3      * ref count. These must be pointers, rather than the actual object or ref count
4      * value, since the goal of a smart pointer is that the reference count is
5      * tracked across multiple smart pointers to one object. */
6     T * obj;
7     unsigned * ref_count;
8   }
```

We know we need constructors and a single destructor for this class, so let's add those first.

```
1   SmartPointer(T * object) {
2     /* We want to set the value of T * obj, and set the reference counter to 1. */
3   }
4
5   SmartPointer(SmartPointer<T>& sptr) {
6     /* This constructor creates a new smart pointer that points to an existing
7      * object. We will need to first set obj and ref_count to pointer to sptr's obj
8      * and ref_count. Then, because we created a new reference to obj, we need to
9      * increment ref_count. */
10  }
11
12  ~SmartPointer(SmartPointer<T> sptr) {
13    /* We are destroying a reference to the object. Decrement ref_count. If
```

```
     * ref_count is 0, then free the memory created by the integer and destroy the
     * object. */
}
```

There's one additional way that references can be created: by setting one `SmartPointer` equal to another. We'll want to override the `equal` operator to handle this, but for now, let's sketch the code like this.

```
onSetEquals(SmartPoint<T> ptr1, SmartPoint<T> ptr2) {
   /* If ptr1 has an existing value, decrement its reference count. Then, copy the
    * pointers to obj and ref_count over. Finally, since we created a new
    * reference, we need to increment ref_count. */
}
```

Getting just the approach, even without filling in the complicated C++ syntax, would count for a lot. Finishing out the code is now just a matter of filling the details.

```
template <class T> class SmartPointer {
 public:
   SmartPointer(T * ptr) {
      ref = ptr;
      ref_count = (unsigned*)malloc(sizeof(unsigned));
      *ref_count = 1;
   }

   SmartPointer(SmartPointer<T> & sptr) {
      ref = sptr.ref;
      ref_count = sptr.ref_count;
      ++(*ref_count);
   }

   /* Override the equal operator, so that when you set one smart pointer equal to
    * another the old smart pointer has its reference count decremented and the new
    * smart pointer has its reference count incrememented. */
   SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
      if (this == &sptr) return *this;

      /* If already assigned to an object, remove one reference. */
      if (*ref_count > 0) {
         remove();
      }

      ref = sptr.ref;
      ref_count = sptr.ref_count;
      ++(*ref_count);
      return *this;
   }

   ~SmartPointer() {
      remove(); // Remove one reference to object.
   }

   T getValue() {
      return *ref;
   }

 protected:
   void remove() {
      --(*ref_count);
      if (*ref_count == 0) {
```