13.2

Compare hash table vs. STL map:

In a hash table, a value is stored by applying hash function on a key. Thus, values are not stored in a hash table in sorted order. Additionally, since hash tables use the key to find the index that will store the value, an insertion or lookup can be done in amortized O(1) time. But we must handle potential collisions in a hash table.

In an STL map, insertion of key/value pair is in sorted order of key. It uses a tree to store values, which is why an O(log N) insert/lookup is required. There is also no need to handle collisions. An STL map works well for things like:

1. find min element

2. find max element

3. print elements in sorted order

4. find the exact element or, if the element is not found, find the next smallest number

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

We can use an STL map. Although this takes O(log n) time, since the number of inputs is small, this time is negligible.

13.3

Virtual functions’ addresses are stored in a “Virtual Table” in C++ class. If any function of a class is declared as virtual, a virtual table is constructed which stores addressed of the virtual functions of this class when the instance of this class is constructed. The compiler would add a hidden vptr variable in the very beginning of all instances of this class which points to the virtual table of the class. If a virtual function is not overridden in the derived class, the virtual table of the derived class stores the address of the function in his parent class. Otherwise, the function with the same name in derived class will override the function in parent class in the virtual table. The v-table is used to resolve the address of the function, for whenever the virtual function is called.

13.4

If we allocate a new area of memory and copy the data from another object to this area of memory to construct a new object with the same type, the whole process is called deep copy. Otherwise if we just copy the pointer or reference from a object to a new object, it’s shadow copy.

Shallow copy is often used when there is a need to pass information about a complex structure without actual duplication of data. Deep copy should be used in most cases, especially when the size of the copied structure is small.

13.5

A variable declared with “volatile” will not be optimized by compiler. Sometimes the compiler will optimize code by eliminating some variable assignment or caching some variable in registers and taking their values always from the register. But these variables might modified by external program and the compiler optimization leads to the changed variable never be read right. The key word “volatile” make the compiler not optimize the code read or write the related variable to ensure the correctness of the program. Volatile variables are also useful when multithreaded programs have global variables and any thread can modify these shared variables.

13.6

In C++, when you have a class with an overloaded method, and you then extend and override that method, you must override all of the overloaded methods, no matter whether it’s virtual or not.

For example:

1 class FirstClass {

2 public:

3 virtual void MethodA (int);

4 virtual void MethodA (int, int);

5 };

6 void FirstClass::MethodA (int i) {

7 std::cout << “ONE!!\n”;

8 }

9 void FirstClass::MethodA (int i, int j) {

10 std::cout << “TWO!!\n”;

11 }

This is a simple class with two methods (or one overloaded method). If you want to override the one-parameter version, you can do the following:

1 class SecondClass : public FirstClass {

2 public:

3 void MethodA (int);

4 };

5 void SecondClass::MethodA (int i) {

6 std::cout << “THREE!!\n”;

7 }

8 void main () {

9 SecondClass a;

10 a.MethodA (1);

11 a.MethodA (1, 1);

12 }

However, the second call won’t work, since the two-parameter MethodA is not visible. That is name hiding.

13.7

Calling a method with an object pointer always invokes:

» the most derived class function, if a method is virtual

» the function implementation corresponding to the object pointer type (used to call the method), if a method is non-virtual.

A virtual destructor works in the same way. A destructor gets called when an object goes out of scope or when we call delete on an object pointer. When any derived class object goes out of scope, the destructor of that derived class gets called first. It then calls its parent class destructor so memory allocated to the object is properly released.

But, if we call delete on a base pointer which points to a derived class object, the base class destructor gets called first (for non-virtual function).

For example:

1 class Base {

2 public:

3 Base() { cout << “Base Constructor “ << endl; }

4 ~Base() { cout << “Base Destructor “ << endl; } /\* see below \*/

5 };

6 class Derived: public Base {

7 public:

8 Derived() { cout << ”Derived Constructor “ << endl; }

9 ~Derived() { cout << ”Derived Destructor “ << endl; }

10 };

11 void main() {

12 Base \*p = new Derived();

13 delete p;

14 }

Output:

Base Constructor

Derived Constructor

Base Destructor

If we declare the base class destructor as virtual, this makes all the derived class destructors virtual as well.

If we replace the above destructor with:

1 virtual ~Base() {

2 cout << “Base Destructor” << endl;

3 }

Then the output becomes:

Base Constructor

Derived Constructor

Derived Destructor

Base Dest ructor

So we should use virtual destructors if we call delete on a base class pointer which points to a derived class.