Algorithm Tutorials

The Best Questions for Would-be C++ Programmers, Part 1

Archive
Normal view
Discuss this article
Write for TopCoder

By **zmij**
TopCoder Member

It seems that an almost obligatory and very important part of the recruitment process is "the test." "The test" can provide information both for the interviewer and the candidate. The interviewer is provided with a means to test the candidate's practical know-how and particular programming language understanding; the candidate can get an indication of the technologies used for the job and the level of proficiency the company expects and even decide if he still wants (and is ready for) the job in question.

I've had my fair share of interviews, more or less successful, and I would like to share with you my experience regarding some questions I had to face. I also asked for feedback from three of the top rated TopCoder members: **bmerry**, **kyky** and **sql_lall** , who were kind enough to correct me where I was not accurate (or plain wrong) and suggest some other questions (that I tried to answer to my best knowledge). Of course every question might have alternative answers that I did not manage to cover, but I tried to maintain the dynamics of an interview and to let you also dwell on them (certainly I'm not the man that knows all the answers). So, pencils up everyone and "let's identify potential C++ programmers or C++ programmers with potential."

1. **What is a class?**
   - A **class** is a way of encapsulating data, defining abstract data types along with initialization conditions and operations allowed on that data; a way of hiding the implementation (hiding the guts & exposing the skin); a way of sharing behavior and characteristics
2. **What are the differences between a C struct and a C++ struct?**
   - A C **struct** is just a way of combining data together; it only has characteristics (the data) and does not include behavior (functions may use the structure but are not tied up to it)
   - Typedefed names are not automatically generated for C structure tags; e.g.,:

```
// a C struct
struct my_struct {
    int someInt;
    char* someString;
};

// you declare a variable of type my_struct in C
struct my_struct someStructure;

// in C you have to typedef the name to easily
// declare the variable
typedef my_struct MyStruct;
MyStruct someOtherStuct;

// a C++ struct
struct MyCppStruct {
    int someInt;
    char* someString;
};
```

```
// you declare a variable of type MyCppStruct in C++
MyCppStruct someCppStruct;
// as you can see the name is automatically typedefed
```

- But what's more important is that a C **struct** does not provide enablement for OOP concepts like encapsulation or polymorphism. Also **"C structs can't have static members or member functions"**, [**bmerry**]. A C++ **struct** is actually a **class**, the difference being that the default member and base class access specifiers are different: **class** defaults to private whereas **struct** defaults to public.

3. **What does the keyword const mean and what are its advantages over #define?**
   - In short and by far not complete, **const** means **"read-only"**! A named constant (declared with **const**) it's like a normal variable, except that its value cannot be changed. Any data type, user-defined or built-in, may be defined as a **const**, e.g.,:

```
// myInt is a constant (read-only) integer
const int myInt = 26;

// same as the above (just to illustrate const is
// right and also left associative)
int const myInt = 26;

// a pointer to a constant instance of custom
// type MyClass
const MyClass* myObject = new MyObject();

// a constant pointer to an instance of custom
// type MyClass
MyClass* const myObject = new MyObject();

// myInt is a constant pointer to a constant integer
const int someInt = 26;
const int* const myInt = &someInt;
```

   - **#define** is error prone as it is not enforced by the compiler like **const** is. It merely declares a substitution that the preprocessor will perform without any checks; that is **const** ensures the correct type is used, whereas **#define** does not. "Defines" are harder to debug as they are not placed in the symbol table.
   - A constant has a scope in C++, just like a regular variable, as opposed to "defined" names that are globally available and may clash. A constant must also be defined at the point of declaration (must have a value) whereas "defines" can be "empty."
   - Code that uses **const** is inherently protected by the compiler against inadvertent changes: e.g., to a class' internal state (**const** member variables cannot be altered, **const** member functions do not alter the class state); to parameters being used in methods (**const** arguments do not have their values changed within methods) [**sql_lall**]. A named constant is also subject for compiler optimizations.
   - In conclusion, you will have fewer bugs and headaches by preferring **const** to **#define**.

4. **Can you explain the private, public and protected access specifiers?**
   - **public**: member variables and methods with this access specifier can be directly accessed from outside the class
   - **private**: member variables and methods with this access specifier cannot be directly accessed from outside the class
   - **protected**: member variables and methods with this access specifier cannot be directly accessed from outside the class with the exception of child classes
   - These access specifiers are also used in inheritance (that's a whole other story, see next question). You can inherit publicly, privately or protected (though I must confess, I cannot see the benefits of the latter).

5. **Could you explain public and private inheritance?[kyky, sql_lall]**

- Public inheritance is the "default" inheritance mechanism in C++ and it is realized by specifying the public keyword before the base class

  ```
  class B : public A
  {
  };
  ```

- Private inheritance is realized by specifying the private keyword before the base class or omitting it completely, as private is the default specifier in C++

  ```
  class B : private A
  {
  };
  or
  class B : A
  {
  };
  ```

- The public keyword in the inheritance syntax means that the publicly/protected/privately accessible members inherited from the base class stay public/protected/private in the derived class; in other words, the members maintain their access specifiers. The private keyword in the inheritance syntax means that all the base class members, regardless of their access specifiers, become private in the derived class; in other words, private inheritance degrades the access of the base class' members - you won't be able to access public members of the base class through the derived one (in other languages, e.g., Java, the compiler won't let you do such a thing).
- From the relationship between the base and derived class point of view,

  ```
  class B : public A {}; B "is a" A but class B : private A {};
  ```

  means B **"is implemented in terms of"** A.
- Public inheritance creates subtypes of the base type. If we have class B : public A {}; then any B object is substituteable by its base calls object (through means of pointers and references) so you can safely write

  ```
  A* aPointer = new B();
  ```

  Private inheritance, on the other hand, class B : private A {};, does not create subtypes making the base type inaccessible and is a form of object composition. The following illustrates that:

  ```
  class A
  {
  public:
      A();
      ~A();
      void doSomething();
  };

  void A :: doSomething()
  {

  }

  class B : private A
  {
  public:
      B();
      ~B();
  ```

```
};
B* beePointer = new B();

// ERROR! compiler complains that the
// method is not accessible
beePointer->doSomething();
// ERROR! compiler complains that the
// conversion from B* to A* exists but
// is not accessible
A* aPointer = new B();
// ! for the following two the standard
// stipulates the behavior as undefined;
// the compiler should generate an error at least
// for the first one saying that B is not a
// polymorphic type
A* aPointer2 = dynamic_cast<A*>(beePointer);
A* aPointer3 = reinterpret_cast<A*>(beePointer);
```

6. **Is the "friend" keyword really your friend?[sql_lall]**
   - The **friend** keyword provides a convenient way to let specific nonmember functions or classes to access the private members of a class
   - friends are part of the class interface and may appear anywhere in the class (class access specifiers do not apply to friends); friends must be explicitly declared in the declaration of the class; e.g., :

```
class Friendly;

// class that will be the friend
class Friend
{
public:
    void doTheDew(Friendly& obj);
};

class Friendly
{
    // friends: class  and function; may appear
    // anywhere but it's
    // better to group them toghether;
    // the default private access specifier does
    // not affect friends
    friend class Friend;
    friend void friendAction(Friendly& obj);
public:
    Friendly(){ };
    ~Friendly(){ };
private:
    int friendlyInt;
};

// the methods in this class can access
// private members of the class that declared
// to accept this one as a friend
void Friend :: doTheDew(Friendly& obj) {
    obj.friendlyInt = 1;
}
```

```
// notice how the friend function is defined
// as any regular function
void friendAction(Friendly& obj)
{
    // access the private member
    if(1 == obj.friendlyInt)
    {
        obj.friendlyInt++;
    } else {
        obj.friendlyInt = 1;
    }
}
```

- "friendship isn't inherited, transitive or reciprocal," that is, your father's best friend isn't your friend; your best friend's friend isn't necessarily yours; if you consider me your friend, I do not necessarily consider you my friend.
- Friends provide some degree of freedom in a class' interface design. Also in some situations friends are syntactically better, e.g., operator overloading - binary infix arithmetic operators, a function that implements a set of calculations (same algorithm) for two related classes, depending on both (instead of duplicating the code, the function is declared a friend of both; classic example is Matrix * Vector multiplication).
- And to really answer the question, yes, **friend** keyword is indeed our friend but always "prefer member functions over nonmembers for operations that need access to the representation."[Stroustrup]

7. **For a class MyFancyClass { }; what default methods will the compiler generate?**
   - The default constructor with no parameters
   - The destructor
   - The copy constructor and the assignment operator
   - All of those generated methods will be generated with the **public** access specifier
   - E.g. MyFancyClass{ }; would be equivalent to the following :

```
class MyFancyClass
{
public:
    // default constructor
    MyFancyClass();
    // copy constructor
    MyFancyClass(const MyFancyClass&);
    // destructor
    ~MyFancyClass();


    // assignment operator
    MyFancyClass& operator=(const MyFancyClass&);
};
```

   - All of these methods are generated only if needed
   - The default copy constructor and assignment operator perform **memberwise** copy construction/assignment of the non-static data members of the class; if references or constant data members are involved in the definition of the class the assignment operator is not generated (you would have to define and declare your own, if you want your objects to be assignable)
   - I was living under the impression that the unary & (address of operator) is as any built-in operator - works for built-in types; why should the built-in operator know how to take the address of your home-brewed type? I thought that there's no coincidence that the "&" operator is also available for overloading (as are +, -, >, < etc.) and it's true is not so common to overload it, as you can live with the

one generated by the compiler that looks like the following:

```
inline SomeClass* SomeClass::operator&()
{
    return this;
}
```

Thanks to **bmerry** for making me doubt what seemed the obvious. I found out the following:

From the ISO C++ standard:
Clause 13.5/6 [over.oper] states that operator =, (unary) & and , (comma) have a predefined meaning for each type. Clause 5.3.1/2 [expr.unary.op] describes the meaning of the address-of operator. No special provisions are mode for class-type objects (unlike in the description of the assignment expression). Clause 12/1 [special] lists all the special member functions, and states that these will be implicitly declared if needed. The address-of operator is not in the list.
From Stroustrup's The C++ Programming Language - Special 3rd Edition:
"Because of historical accident, the operators = (assignment), & (address-of), and , (sequencing) have predefined meanings when applied to class objects. These predefined meanings can be made inaccessible to general users by making them private:... Alternatively, they can be given new meanings by suitable definitions."
From the second edition of Meyer's Effective C++:
"A class declaring no operator& function(s) does NOT have them implicitly declared. Rather, compilers use the built-in address-of operator whenever "&" is applied to an object of that type. This behavior, in turn, is technically not an application of a global operator& function. Rather, it is a use of a built-in operator." In the errata http://www.aristeia.com/BookErrata/ec++2e-errata_frames.html

8. **How can you force the compiler not to generate the above mentioned methods?**
   - Declare and define them yourself - the ones that make sense in your class' context. The default no-parameters constructor will not be generated if the class has at least one constructor with parameters declared and defined.
   - Declare them private - disallow calls from the outside of the class and DO NOT define them (do not provide method bodies for them) - disallow calls from member and friend functions; such a call will generate a linker error.
9. **What is a constructor initialization list?**
   - A **special** initialization point in a constructor of a class (initially developed for use in inheritance).
   - Occurs only in the definition of the constructor and is a list of constructor calls separated by commas.
   - The initialization the constructor initialization list performs occurs before any of the constructor's code is executed - very important point, as you'll have access to fully constructed member variables in the constructor!
   - For example:

```
// a simple base class just for illustration purposes
class SimpleBase
{
public:
    SimpleBase(string&);
    ~SimpleBase();
private:
    string& m_name;
};

// example of initializer list with a call to the
// data member constructor
SimpleBase :: SimpleBase(string& name) : m_name(name)
{
}

// a class publicly derived from SimpleBase just for
// illustration purposes
class MoreComplex : public SimpleBase
{
public:
    MoreComplex(string&, vector<int>*, long);
    ~MoreComplex();
private:
    vector<int>* m_data;
    const long m_counter;
};


// example of initializer list with calls to the base
// class constructor and data member constructor;
// you can see that built-in types can also be
// constructed here
MoreComplex :: MoreComplex(string &name,
    vector<int>* someData, long counter) :
    SimpleBase(name), m_data(someData),
    m_counter(counter)
{

}
```

- As you saw in the above example, built-in types can also be constructed as part of the constructor initialization list.
- Of course you do not have to use the initialization list all the time (see the next question for situations that absolutely require an initialization list) and there are situations that are not suitable for that: e.g., you have to test one of the constructor's arguments before assigning it to your internal member variable and throw if not appropriate.
- It is recommended that the initialization list has a consistent form: first the call to the base class(es) constructor(s), and then calls to constructors of data members in the order they were specified in the class' declaration . Note that this is just a matter of coding style: you declare your member variables in a certain order and it will look good and consistent to initialize them in the same order in the initialization list.

10. **When "must" you use a constructor initialization list?**
    - **Constant** and **reference** data members of a class may only be initialized, never assigned, so you **must** use a constructor initialization list to properly construct (initialize) those members.
    - In inheritance, when the base class does not have a default constructor or you want to change a default argument in a default constructor, you have to explicitly call the base class' constructor in the initialization list.
    - For reasons of correctness - any calls you make to member functions of sub-objects (used in composition) go to initialized objects.
    - For reasons of efficiency. Looking at the previous question example we could rewrite the SimpleBase constructor as follows:

```
SimpleBase :: SimpleBase(string &name)
{
    m_name = name;
}
```

The above will generate a call to the default string constructor to construct the class member m_name and then the assignment operator of the string class to assign the name argument to the m_name member. So you will end up with two calls before the data member m_name is fully constructed and initialized.

```
SimpleBase :: SimpleBase(string &name) : m_name(name)
{

}
```

The above will only generate a single call, which is to the copy constructor of the string class, thus being more efficient.

That's it for the first part of this installment. Stay tuned for the second one, as we're going to go deeper into the language features. Good luck on those interviews!

References

[1] - Bjarne Stroustrup - The C++ Programming Language Special 3rd Edition
[2] - Stanley B. Lippman, Josee Lajoie, Barbara E. Moo - C++ Primer
[3] - C++ FAQ Lite
[4] Herb Sutter - Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions
[5] Scott Meyers - Effective C++: 55 Specific Ways to Improve Your Programs and Designs
[6] Scott Meyers - More Effective C++: 35 New Ways to Improve Your Programs and Designs
[7] Bruce Eckel - Thinking in C++, Volume 1: Introduction to Standard C++