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

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;
}
```

8. Thanks to **bmerry** for making me doubt what seemed the obvious. I found out the following:
9. **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 erratahttp://www.aristeia.com/BookErrata/ec++2e-errata_frames.html

10. **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.
11. **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.
12. **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;
}
```

13. 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.

```
14.  SimpleBase :: SimpleBase(string &name) : m_name(name)
15.  {
16.
17.  }
```

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

In the second part of this installment we'll tackle some questions regarding more advanced features of the language (the experienced C++ programmers will consider some of these more on the basic side). So let's get to it and work on the second part of this "interview".

1. **What are virtual functions?**
   - Virtual functions represent the mechanism through which C++ implements the OO concept of polymorphism. Virtual functions allow the programmer to redefine in each derived class functions from the base class with altered behavior so that you can call the right function for the right object (allow to perform the right operation for an object through only a pointer/reference to that object's base class)
   - A member function is declared virtual by preceding its declaration (not the definition) with the **virtual** keyword

```
class Shape
{
public:
    ...
    //a shape can be drawn in many ways
    virtual void draw(){ };
};
```

"A virtual function must be defined for the class in which it is first declared ..." [Stroustrup]. The redefinition of a virtual function in a derived class is called **overriding**(complete rewrite) or **augmentation** (rewrite but with a call to the base class function)

```
class Rectangle : public Shape
{
public:
    ...
    void draw() { };
};

class Square : public Rectangle
{
public:
    ...
    void draw() { };
};
    ...

Shape* theShape = new Square();
// with the help of virtual functions
// a Square will be drawn and not
// a Rectangle or any other Shape
theShape->draw();
```

   - Through virtual functions C++ achieves what is called **late binding** (dynamic binding or runtime binding), that is actually connecting a function call to a function body at runtime based on the type of the object and not at compilation time (static binding) (**)
2. **What is a virtual destructor and when would you use one?**
   - A virtual destructor is a class' destructor conforming to the C++'s polymorphism mechanism; by declaring the destructor virtual you ensure it is placed in the **VTABLE**of the class and it will be called at proper times
   - You make a class' destructor virtual to ensure proper clean-up when the class is supposed to be subclassed to form a hierarchy and you want to delete a derived object thorough a pointer to it (the base class)
   - E.g. :

```
#include <vector>
#include <iostream>

using namespace std;
```

```cpp

class Base
{
public:
    Base(const char* name);
    // warning! the destructor should be virtual
    ~Base();

    virtual void doStuff();
private:
    const char* m_name;
};

Base :: Base(const char* name) : m_name(name)
{

}

Base :: ~Base()
{

}



void Base :: doStuff()
{
    cout << "Doing stuff in Base" << endl;
}



class Derived : public Base
{
public:
    Derived(const char* name);
    ~Derived();

virtual void doStuff();
private:
    vector<int>* m_charCodes;
};

Derived :: Derived(const char* name) : Base(name)
{
    m_charCodes = new vector<int>;
}

Derived :: ~Derived()
{
    delete m_charCodes;
}

void Derived :: doStuff()
{
    cout << "Doing stuff in Derived" << endl;
}

int main(int argc, char* argv[])
{
    // assign the derived class object pointer to
    // the base class pointer
    char* theName = "Some fancy name";
    Base* b = new Derived(theName);

    // do some computations and then delete the
    // pointer
    delete b;
    return 0;
}
```

What will happen in our rather lengthy example? Everything seems OK and most of the available C++ compilers will not complain about anything **(*)**. Nevertheless there is something pretty wrong here. The C++ standard is clear on this topic: **when you want to delete a derived class object through a base class pointer and the destructor of the base class is not virtual the result is undefined**. That means you're on your own from there and the compiler won't help you! What is the most often behavior in such situations is that the derived class' destructor is never called and parts of your derived object are left undestroyed. In the example above you will leave behind a memory leak, the m_charCodes member will not be destroyed because the destructor ~Derived() will not be called

- A thing to notice is that declaring all destructors virtual is also pretty inefficient and not advisable. That makes sense (declaring the destructor virtual) only if your class is supposed to be part of a hierarchy as a base class, otherwise you'll just waste memory with the class' **vtable** generated only for the destructor. So declare a virtual destructor in a class **"if and only if that class is part of a class hierarchy, containing at least one virtual function. In other words, it is not necessary for the class itself to have that virtual function - it is sufficient for one of its descendents to have one."[kyky]**

3. **How do you implement something along the lines of Java interfaces in C++?[kyky]**
   - C++ as a language does not support the concept of "interfaces" (as opposed to other languages like Java or D for example), but it achieves something similar through **Abstract Classes**
   - You obtain an abstract class in C++ by declaring at least one **pure virtual function** in that class. A virtual function is transformed in a pure virtual with the help of the initializer **"= 0"**. A pure virtual function does not need a definition. An abstract class cannot be instantiated but only used as a base in a hierarchy

```
class MySillyAbstract
{
public:
    // just declared not defined
    virtual void beSilly() = 0;
};
```

A derivation from an abstract class must implement all the pure virtuals, otherwise it transforms itself into an abstract class

- You can obtain an **"interface"** in C++ by declaring an abstract class with all the functions pure virtual functions and public and no member variables - only behavior and no data

```
class IOInterface
{
public:
    virtual int open(int opt) = 0;
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
};
```

[adapted after an example found in Stroustup The C++ Programming Language 3rd Edition]
In this way you can specify and manipulate a variety of IO devices through the interface.

4. **Could you point out some differences between pointers and references?**
   - A reference must always be initialized because the object it refers to already exists; a pointer can be left uninitialized (though is not recommended)
   - There's no such thing a s a **"NULL reference"** because a reference must always refer to some object, so the "no object" concept makes no sense; pointers, as we all know, can be **NULL**
   - References can be more efficient, because there's no need to test the validity of a reference before using it (see above comment); pointers most often have to be tested against **NULL** to ensure there are valid objects behind them
   - Pointers may be reassigned to point to different objects (except constant pointers, of course), but references cannot be (references are "like" constant pointers that are automatically dereferenced by the compiler)
   - References are tied to someone else's storage (memory) while pointers have their own storage they account for
   - One would use the dot operator **"."** to access members of references to objects, however to access members of pointers to objects one uses the arrow **"->"[sql_lall]**
5. **When would you use a reference?**
   - You should use a **reference** when you certainly know you have something to refer to, when you never want to refer to anything else and when implementing operators whose syntactic requirements make the use of pointers undesirable; in all other cases, "stick with pointers"
   - Do not use references just to reduce typing. That (and that being the sole reason) is not an appropriate usage of the reference concept in C++; using references having in mind just the reason of reduced typing would lead you to a "reference spree" - it must be clear in one's mind when to use references and when to use pointers; overusing any of the two is an inefficient path
6. **Can you point out some differences between new & malloc?**
   - **"new"** is an operator built-in into the C++ language, **"malloc"** is a function of the C standard library
   - **"new"** is aware of constructors/destructors, **"malloc"** is not; e.g. :

```
string* array1 = static_cast<string*>(malloc(10 * sizeof(string)));
free(array1);
```

array1 in the above example points to enough memory to hold 10 strings but no objects have been constructed and there's no easy and clean (proper) way from OO point of view to initialize them (see the question about **placement new** - in most day to day programming tasks there's no need to use such techniques). The call to**free()** deallocates the memory but does not destroy the objects (supposing you managed to initialize them).

```
string* array2 = new string[10];
delete[] array2;
```

on the other hand **array2** points to 10 fully constructed objects (they have not been properly initialized but they are constructed), because **"new"** allocates memory and also calls the **string** default constructor for each object. The call to the **delete** operator deallocates the memory and also destroys the objects

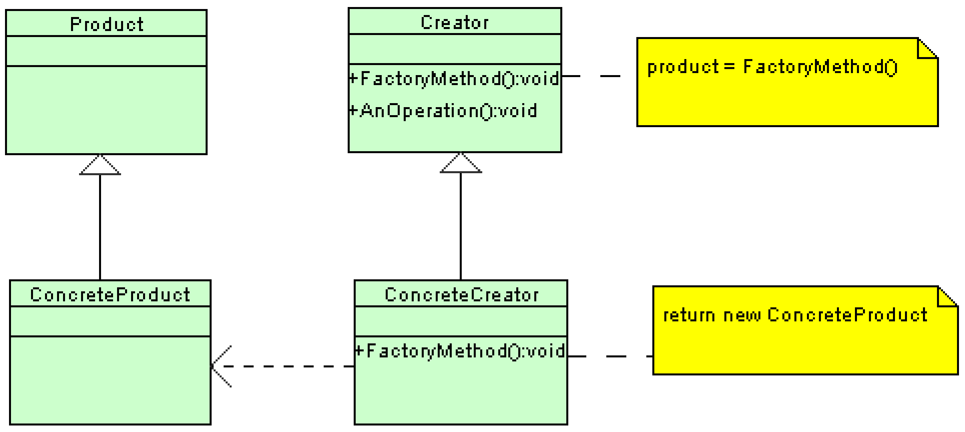
- You got to remember to always use **free()** to release memory allocated with **malloc()** and **delete** (or the array correspondent **delete[]**) to release memory allocated with **new** (or the array correspondent **new[]**)

7. **What are the differences between "operator new" and the "new" operator?**
   - **"new"** is an operator built into the language and it's meaning cannot be changed; **"operator new"** is a function and manages **how** the **"new"** operator allocates memory its signature being: **void* operator new(size_t size)**
   - The **"new"** operator is allowed to call a constructor, because **new** has 2 major steps in achieving its goals : in step 1 it allocates enough memory using **"operator new"** and then in step 2 calls the constructor(s) to construct the object(s) in the memory that was allocated
   - **"operator new"** can be overridden meaning that you can change the way the **"new"** operator allocates memory, that is the mechanism, but not the way the **"new"**operator behaves, that is it's policy(semantics) , because what **"new"** does is fixed by the language
8. **What is "placement new"?**
   - A special form of constructing an object in a given allocated zone of memory
   - The caller already knows what the pointer to the memory should be, because it knows where is supposed to be placed. **"placement new"** returns the pointer that's passed into it
   - Usage of **"placement new"** implies an explicit call to the object's destructor when the object is to be deleted, because the memory was allocated/obtained by other means than the standard **"new"** operator allocation
   - E.g. :

```
// supposing a "buffer" of memory large enough for
// the object we want to construct was
// previously allocated using malloc
MyClass* myObject = new (buffer) MyClass(string& name);


// !!ERROR
delete myObject;
// the correct way is
myObject->~MyClass();
// then the "buffer" must also be properly
// deallocated
free(buffer);
```

9. **What is a "virtual constructor"?[kyky]**
   - There is no such thing as a virtual constructor in C++ simply because you need to know the exact type of the object you want to create and virtual represent the exact opposite concept **(***)**
   - But using an indirect way to create objects represents what is known as **"Virtual Constructor Idiom"**. For example you could implement a **clone()** function as an indirect copy constructor or a **create()** member function as an indirect default constructor (C++ FAQ Lite)
   - The GoF calls a variant of this idiom the Factory Method Pattern - "define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses". A concrete example will speak for itself:



[Created using the TopCoder UML Tool]

```
// Product
class Page
{
};
```

```cpp
// ConcreteProduct
class SkillsPage : public Page
{
};

// ConcreteProduct
class ExperiencePage : public Page
{
};

// ConcreteProduct
class IntroductionPage : public Page
{
};

// ConcreteProduct
class TableOfContentsPage : public Page
{
};

// Creator
class Document
{
// Constructor calls abstract Factory method
public:
    Document();

    // Factory Method
    virtual void CreatePages() { };
protected:
    std::list<Page*> thePageList;
};

Document :: Document()
{
    CreatePages();
};

// ConcreteCreator
class Resume : public Document
{
public:
    // Factory Method implementation
    void CreatePages();
};

// Factory Method implementation
void Resume :: CreatePages()
{
    thePageList.push_back(new SkillsPage());
    thePageList.push_back(new ExperiencePage());
}

// ConcreteCreator
class Report : public Document
{
public:
    // Factory Method implementation
    void CreatePages();
};

// Factory Method implementation
void Report :: CreatePages()
{
    thePageList.push_back(new TableOfContentsPage());
    thePageList.push_back(new IntroductionPage());
}
```

```
int main(int argc, char* argv[])
{
    // Note: constructors call Factory Method
    vector<Document*> documents(2);
    documents[0] = new Resume();
    documents[1] = new Report();

    return 0;
}
```

10. **What is RAII?**

- **RAII** - **R**esource **A**cquisition **I**s **I**nitialization - is a C++ technique (but not limited to the C++ language) that combines acquisition and release of resources with initialization and uninitialization of variables
- E.g. :

```
// this is a hypothetic LogFile class using an
// hypothetic File class just for the illustration
// of the technique
class LogFile
{
public:
    LogFile(const char*);
    ~LogFile();

    void write(const char*);
private:
    File* m_file;
};

LogFile :: LogFile(const char* fileName) :
// ! acquisition and initialization
m_file(OpenFile(fileName))
{
    if(NULL == m_file)
    {
        throw FailedOpenException();
    }
}

LogFile :: ~LogFile()
{
    // ! release and uninitialization
    CloseFile(m_file);
}

void LogFile :: write(const char* logLine)
{
    WriteFile(m_file, logLine);
}

// a hypothetical usage example
void SomeClass :: someMethod()
{
    LogFile log("log.tx");
    log.write("I've been logged!");

    // !exceptions can be thrown without
    // worrying about closing the log file
    // or leaking the file resource
    if(...)
    {
    throw SomeException();
    }
}
```

- Without **RAII** each usage of the LogFile class would be also combined with the explicit management of the File resource. Also in the presence of exceptions you would have to be careful and clean-up after yourself, thing that is taken care of with the proper usage of **RAII** as illustrated in the example above

- **RAII** is best used with languages that call the destructor for local objects when they go out of scope (implicit support of the technique) like C++. In other languages, like Java & C#, that rely on the garbage collector to destruct local objects, you need finalization routines (e.g. try-finally blocks) to properly use **RAII**
- Real usage examples of the technique are the C++ Standard Library's file streams classes and STL's auto_ptr class (to name just very, very few)

That was it, folks! I hope that even if those questions did not pose any challenges, you still had fun doing/reading this quiz and refreshing your memory on some aspects of the C++ language. Good luck on those interviews!

Notes

(*) **bmerry** suggested that my claim is not accurate but I've tested the example on Windows XP: Visual Studio 2005 Professional Edition (the evaluation one that you can get from the Microsoft site) did not warn, not even after setting the warnings level to Level 4 (Level 3 is the default one); Mingw compiler based on GCC (that comes with theBloodshed DevCpp version 4.9.9.2) also did not warn (the compiler settings from within the IDE are minimalist; tried to pass -pedantic and -Wextra to the compiler command line but still no success); Digital Mars C++ compiler (dmc) also did not warn with all warnings turned on; Code Warrior Professional Edition 9 does not warn also (this is pretty old, but Metrowerks compilers were renowned for the robustness and standard conformance). So, unless you start digging through the documentation of those compilers to find that right command line switch or start writing the right code, you're in the harms way at least with the "out of the box" installations of these compilers.

(**) The compiler does all the magic: first, for each class that contains virtual functions (base and derived), the compiler creates a static table called the **VTABLE**. Each virtual function will have a corresponding entry in that table (a function pointer); for the derived classes the entries will contain the overridden virtual functions' pointers. For **each base**class (it's not static, each object will have it) the compiler adds a hidden pointer called the **VPTR**, that will be initialized to point to the beginning of the **VTABLE** - in the derived classes the (same) **VPTR** will be initialized to point to the beginning of the derived class' **VTABLE**. So when "you make a virtual function call through a base class pointer (that is, when you make a polymorphic call), the compiler quietly inserts code to fetch the **VPTR** and look up the function address in the **VTABLE**, thus calling the correct function". This might seem overly complicated but on a typical machine it does not take much space and it's very, very fast as a smart man said once "fetch, fetch call".

(***) For that and other fine C++ gems go to Stroustrup.

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] Gamma, Helm, Johnson, Vlissides (GoF) - Design Patterns Elements of Reusable Object-Oriented Software
[5] Herb Sutter - Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions
[6] Scott Meyers - Effective C++: 55 Specific Ways to Improve Your Programs and Designs
[7] Scott Meyers - More Effective C++: 35 New Ways to Improve Your Programs and Designs
[8] Bruce Eckel - Thinking in C++, Volume 1: Introduction to Standard C++