C++ Review “Quiz” Answers

Leor Zolman, BD Software

Updated 8/20/2011

1) How many distinct contexts are there in C++ for the use of the **static** keyword?

Answer: 4. Two from C, two added to C++.

From C:

1. Block scope static data and functions:

```
void f() {
    int i;              // automatic
    static int j = 10;  // static storage (fixed location)
                        // C: initialized at compile time
                        // C++: Initialized 1st time thru
}
```

2. File scope static, gives name *internal linkage* (deprecated for C++):

```
// Top level of file
static int secret;    // invisible outside translation unit
static void f() { … } // Ditto
```

From C++:

1. static data members: one copy *per class*
2. static member functions: have no *this*

2) What does the term *static storage* mean? (In this case, the word *static* is *not* being used as a language keyword)

Answer (From an online IBM glossary):

“An area that is allocated by the system when a program is activated. Static storage exists as long as the program activation exists. If the program has not been deactivated, the values in the storage persist from one call to another.” In other words, fixed storage throughout the lifetime of the running app.

3) What are the C++ “storage classes” ?

Answer:

1. Automatic (non-static block scope objects, typically residing on the stack)
2. Static (as above)
3. External (name visible across translation units)
4. Register (Not used much any more…)

NOTE: "Dynamic memory" is NOT a storage class! That's because storage classes apply only to *names* of objects, and memory obtained via `new`/`malloc` is *unnamed*.

4) What's the difference between *internal linkage* and *external linkage?*

Answer: Objects with names having external linkage are visible to other translation units; names with internal linkage are not even identified in the object file, so therefore they can never be referred to from outside the translation unit where they are defined.

5) In the class hierarchy below, is **~D** (class **D**'s destructor) virtual?

```
class B                             class D : public B
   {                                {
public:                               public:
    B();                                 D();
    virtual ~B();                        int f(double x);
    int f(double x);                     int f();
    virtual void g(int i);               void g(int i);
    virtual void g();               };
 };

int main() {
    B *bp = new D;
    bp->f(2.23);     // Call #1
    bp->g(10);       // Call #2
    bp->f();         // Call #3

    D *dp = new D;
    dp->g();         // Call #4
};
```

Answer: Yes, because **B**'s destructor is virtual

6) Does **D::f(double)** override **B::f(double)** ? If not, what *does* it do?

Answer: No, because **B::f(double)** is non-virtual. The term "override" only applies to the redefinition of *virtual* functions inherited from a base class. We can say that **D::f(double)** *hides* or *redefines* **B::f(double)**.

7) For the calls marked #1-#4: Legal? If so, what gets called?

Answer:

Call #1: Legal, but bad design, you should *never* redefine a non-virtual function!
Calls **B::f(double)** (is this something you really want to see happen?)

Call #2: Legal and correct. Calls **D::g(int).**

Call #3: Illegal (will not compile). The "static type" of the pointer must support the function signature applied in a function call. Only *then* would virtual dispatch mechanisms kick in.

Call #4: Illegal (will not compile). The definition of **D::g(int)** *hides* all declarations of **g** that would otherwise have been inherited from **B**.

8) Is the code below C++ Standard-conformant?

```
#include <iostream>
int main()
{
     double vals[] = { 1.1, 2.2, 3.3, 4.4};
     std::cout << vals[1] << std::endl; // 2.2
     std::cout << 1[vals] << std::endl; // 2.2
     double *vp = vals + 1;
     std::cout << vp[0] << std::endl;   // 2.2
     std::cout << 0[vp] << std::endl;   // 2.2
}            // (Aside: anything missing here?
```

Answer: Yes, completely. This is *pointer-array duality.* Two important things to note:

Any array name **a** "decays" when used in a run-time expression to behave as if it had been written **&a[0]**

Any valid expression of form

a[b]

involving an array name and an integral expression (one of them is a, the other one b) can be re-written as:

***(a + b)**

Then, simple commutivity of addition allow the rewrite as:

***(b + a)**

And finally, this can be rewritten as:

b[a]

Hence, *all four* forms are 100% equivalent!
As for the "aside": if **main()** is declared to return int and nothing is returned explicitly, then it behaves as if "return 0" were used.

9) If so, will the output be as the comments indicate?

Answer: Yes, absolutely.

10) Does **vp** conceptually represent a *pointer* or an *array*?

Answer: “Yes” ;-)

11) What does it mean for a program to be “**const** correct” ?

Answer: IMO, when the `const` keyword is used in all places that it makes sense for it to be used (modifying pointer and reference function parameters, and to define **const** member functions).

12) What’s the difference (if any) between the following two function declarations:

```
// (Use this version, not the one from the printed book)
int f(const int x);
int f(int x);
```

Answer: NONE. The const is 100% useless in this context and completely ignored by C++. (Note: in a function *definition*, on the other hand, some people use const like this to “document” that a parameter’s value should not be monkeyed with…)

13) [Guru Question!]: What *implicit* declaration is affected by taking a non-**const** member function and changing its declaration and definition so that it becomes a **const** member function?

Answer: The implicit declaration of **this** for a function in class **T** goes from:

T *const this;

to:

const T * const this;

14) From an OO perspective, what “policy” (with respect to the function in question) is being dictated by a base class to its subclasses in each of the following cases?

- A non-virtual (non-static) member function

  Answer: “You must support the function, and you must accept my implementation.” (Note: *NOT* enforced by the language!!!)

- An impure virtual function

  Answer: “You must support the function, either accepting my default implementation or overriding it to provide your own.”

- A pure virtual function

  Answer: “You must support this function, and you must write your own implementation.”

15) What is meant by the concept of an *Abstract Base Class,* and how do you define one in C++?

Answer: An ABC is a class that is not meant be instantiated, but only to be used as a base. To force a class to be abstract, you must define at least one pure virtual function in the class.

16) What does an ABC *demand* of its derived classes if they wish to be *concrete*?

Answer: For a derived class of an ABC to be concrete (to be able to be instantiated), it must provide its own definitions for each pure virtual function inherited from the base.

17) Is the following C++ code Standard-conformant? Will it compile without error? Why or why not?

```
#include <iostream>
int main()
{
     char *str1 = "ABCDEFG";
     char *str2 = "ABCDEFG";

     str1[3] = '*';

     std::cout << str1 << std::endl;
     std::cout << str2 << std::endl;

}
```

Answer: NOT Standard conformant, because modification of a string literal has *undefined behavior* (string literals have type “**array of const char**”)
However, this code WILL compile without error due to a “grandfather clause” based on early C usage of **char *** variables to represent literal strings (before the **const** keyword was incorporated into the language…stolen from C++, by the way!)

18) [Guru question!]: If it compiles, what are three different possible runtime results?

Answer:

1. Both strings display with the ‘D’ replaced by ‘*’
2. The program crashes due to a segmentation fault or similar issue
3. Only str1 shows the change (behaves as expected)