Static Variables and Static Class Members - 2017

bogotobogo.com site search:

Static Object - Summary

1. Persistence: it remains in memory until the end of the program.
2. File scope: it can be seen only withing a file where it's defined.
3. Visibility: if it is defined within a function/block, it's scope is limited to the function/block. It cannot be accessed outside of the function/block.
4. Class: static members exist as members of the class rather than as an instance in each object of the class. So, **this** keyword is not available in a static member function. Such functions may access only static data members. There is only a single instance of each static data member for the entire class:
   A static data member : class variable
   A non-static data member : instance variable
5. Static member function: it can only access static member data, or other static member functions while non-static member functions can access all data members of the class: static and non-static.

Interview Question - What's static?

This is one of the frequently asked questions during the interview. Most of the candidates get the first one, and some with the second, third, and may be 4th ones as well. But rarely they address the 5th one:

1. A variable declared static within the body of a function maintains its value between invocations of the function.
2. A variable declared static within a module (but outside the body of a function) is accessible by all functions within that module. However, it is not accessible by functions from other modules.
3. static members exist as members of the class rather than as an instance in each object of the class. There is only a single instance of each static data member for the entire class.
4. Non-static member functions can access all data members of the class: static and non-static.
   Static member functions can only operate on the static data members.
5. C functions declared static within a module may only be called by other functions within that module (file scope).

Static Objects

Static object is an object that persists from the time it's constructed until the end of the program. So, **stack** and **heap** objects are excluded. But global objects, objects at namespace scope, objects declared **static** inside classes/functions, and objects declared at file scope are included in **static objects**. Static objects are destroyed when the program stops running.

Static Storage

Variables defined **outside a function** or by using the keyword **static** have static storage duration. They persist for the entire running time of a program.

These variables can be classified as three groups in terms of **linkage**:

1. external linkage
2. internal linkage
3. no linkage

Since the static variables stay the same throughout the life cycle of the program, they are easy to deal with for the memory system and they are allocated in a fixed block of memory.

Here is an example of static variables with different duration.

```
int a= 1;
static int b = 2;
int main() {}
void f() {
        static int c = 3;
        int d = 4;
}
```

All the **static** variables persist until program terminates. The variable **d** has local scope and no linkage - it's no use outside of **f()**. But **c** remains in memory even when the **f()** function is not being executed. By saying that **c** is **static**, we are saying that we want to allocate it once, and only once, at some point before the first time that **f()** is called, and that we do not want to deallocate it as long as our program runs.

Both **a** and **b** can be accessed from the point of declaration until the end of the file. But **a** can be used in other files because it has external linkage.

All static duration variables have the following initialization characteristics:

1. An uninitialized static variable set to **0**.
2. A static variable can be initialized only with a **constant expression**.

```
int x;                // x set to 0
int y = 50;           // 50 is literal constant
int z = sizeof(int); // sizeof ok
int zz = 10 * x;      // not allowed, x is not constant
int main()
{...}
```

Const and Extern

The const in C++ has gives a little bit of twist to the default storage classes. While a global variable has external linkage by default, a **const** global has **internal** linkage by default. In other words, C++ treats a global **const** definition as if the **static** had been used as in the following code.

```
const int a = 10;
int main() { ....
```

So, the "const int a = 10" becomes "static const int a = 10". However, if global **const** had external linkage as regular variables do, the **const** declaration would be an error because we can define a global variable in one file only. In other words, only one file can contain the preceding declaration, and the other files have to provide reference declarations using the **extern** keyword. Note that only the declarations without the **extern** can initialize values.

However, if we want to make a constant have external linkage, we can use the **extern** keyword to override the default internal linkage:

```
extern const int a = 20;
```

Here, we should use the **extern** keyword to declare the constant in all files that use the constant. That's the difference between regular external variables and constant. For regular external variables, we don't use the keyword **extern** when we define a variable, but we use **extern** in other files using that variable.

extern

An **extern** declaration does not define the variable unless it is also initialized in the same statement.

```
extern int x;       // declaration
extern int x = 10;  // definition
```

For **extern "C"**, please visit extern "C".

Static Storage and Dynamic Allocation

Dynamic memory is controlled by the **new** and **delete** operators, **not** by **scope** and **linkage** rules. So, dynamic memory can be allocated from one function and freed from another function.

Although the storage schemes don't apply to dynamic memory, then **do** apply to **automatic** and **static** pointer variables used to keep track of dynamic memory.

Let's look at the following line of code:

```
int *ptr = new int[10];
```

The 40 bytes of memory allocated by **new** remains in memory until the **delete** frees it. But the pointer **ptr** passes from existence when the function containing this declaration terminates.

If we want to have the 40 bytes of allocated memory available from another function, we need to pass or return its address to that function.

On the other hand, if we declare **ptr** with external linkage, the **ptr** pointer will be available by using:

```
extern int *ptr;
```

However, a statement that uses **new** to set **ptr** has to be in a function because static storage variables can only be initialized with constant expressions as shown in the following example.

```
int *ptr;
// initialization with non-const not allowed here
// int *ptr = new int[10];
int main()
{
        ptr = new int[10];
...
```

See Memory Allocation.

Static Class Member

Static members exist as members of the class rather than as an instance in each object of the class. So, **this** keyword is not available in a **static** member function. Such functions may access only **static** data members. There is a single instance of each **static** data member for the entire class, which should

be initialized, usually in the source file that implements the class member functions. Because the member is initialized outside the class definition, we must use fully qualified name when we initialize it:

```
class_name::static_member_name = value;
```

Here is the real life example, Car.h:

```
class Car
{
private:
        static int id;
public:
        Car();
...
};
```

Implementation file, Car.cpp:

```
#include <iostream>

int Car::id = 100;
...
Car::Car() {}
....
```

The code initializes the static **id** member to 100. Note again that we cannot initialize a static member variable inside the class declaration. That's because the declaration is a description of how memory is to be allocated, but it doesn't allocate memory. We allocate and initialize memory by creating an object using that format.

In the case of a static class member, we initialize the static member independently, with a separate statement outside the class declaration. That's because the static class member is stored separately rather than as part of an

object.

The exception to the initialization of a static data member inside the class declaration is if the static data member is a **const** of integral or enumeration type.

```
#include <iostream>

class Car
{
        enum Color {silver = 0, maroon, red };
        int year;
        int mileage = 34289;                    // error: not-static da
                                                // only static const in
                                                // can be initialized w

        static int vin = 12345678;              // error: non-constant
                                                // only static const in
                                                // can be initialized w

        static const string model = "Sonata";   // error: not-integral
                                                // cannot have in-class

        static const int engine = 6;            // allowed: static cons
};

int Car::year = 2013;                           // error: non-static da
                                                // cannot be defined ou

int main()
{
        return 0;
}
```

The following example shows an illegal access to non-static member from a static function.

```
class X
```

```
{
public:
        int x;
        static void f(int);
};

void X::f(int z) {x=z;}
```

In function f(), x=z is an error because f(), a static function, is trying to access non-static member x.

So, the fix should be like this:

```
class X
{
public:
        static int x;
        static void f(int);
};

void X::f(int z) {x=z;}
```

Non-static members can not be used as default arguments

The following code shows that a non-static members can not be used as default arguments.

```
#include <iostream>
int xGlobal = 7;

struct Foo
{
  int xMember;

  static int xStatic;
```

```
  Foo(int x) : xMember(x) {}

  int a(int x = xGlobal)
  {
    return x;
  }

  int b(int x = xMember) // wrong: won't compile
  {
    return x;
  }

  int c(int x = xStatic)
  {
    return x;
  }
};

int Foo::xStatic = 1;

int main()
{
  Foo f(911);

  std::cout << f.a() << std::endl;
  std::cout << f.b() << std::endl;
  std::cout << f.c() << std::endl;

  return 0;
}
```

Static Member Functions

Here are some characteristics of **static member functions**:

1. A static member function can only access **static member data**, **static member functions** and **data and functions outside the class**. So, we

must take note not to use static member function in the same manner as non-static member function, as non-static member function can access all of the above including the static data member.

2. We must first understand the concept of static data while learning the context of static functions. It is possible to declare a data member of a class as static irrespective of it being a public or a private type in class definition. If a data is declared as static, then the static data is created and initialized only once. Non-static data members are created again and again. For each separate object of the class, the static data is created and initialized only once. As in the concept of static data, all objects of the class in static functions share the variables. This applies to all objects of the class.
3. A non-static member function can be called only after instantiating the class as an object. This is not the case with static member functions. A static member function can be called, even when a class is not instantiated.
4. A static member function cannot have access to the this pointer of the class.
5. A non-static member function can be declared as virtual but care must be taken not to declare a static member function as virtual.

Static - Singleton Pattern

Singleton design pattern is a good example of static member function and static member variable. In this pattern, we put constructor in **private** section not in public section of a class. So, we can not access the constructor to make an instance of the class. Instead, we put a public function which is **static** function. The **getInstance()** will make an instance only once. Note

that if this method is not static, there is no way to invoke the **getInstance()** even though it is public method. That's because we do not have any instance of **Singleton**.

```
#include <iostream>

using namespace std;

class Singleton
{
public:
        static Singleton *getInstance();
private:
        Singleton() {}
        static Singleton *instance;
};

Singleton* Singleton::instance = 0;
Singleton* Singleton::getInstance() {
                if(!instance) {
                        instance = new Singleton();
                        cout << "getInstance(): First instance\n";
                        return instance;
                }
                else {
                        cout << "getInstance(): previous instance\n";
                        return instance;
                }
}

int main()
{
        Singleton* s1 = Singleton::getInstance();
        Singleton* s2 = Singleton::getInstance();

        return 0;
}
```

Output is:

```
getInstance(): First instance
getInstance(): previous instance
```

External Linkage and static keyword

External linkage means that a symbol in one translation unit can be accesses from the other translation units. Unless we take special steps, the functions and global variables in our **.cpp** will have external linkage.

```
const int MY_CONSTANT = 199;     // (note)In C++, this has internal li
std::string MY_NAME = "BoGo";
void My_Function() {}
```

Here is the code that can access the **My_CONSTANT**, **My_NAME**, and **My_Function()** without using public interfaces, thus breaking encapsulation:

```
extern const int MY_CONSTANT;
extern std::string MY_NAME;
extern void MY_Function();
....
```

One way of avoid this linkage issue is using **static** keyword:

```
static const int MY_CONSTANT = 199;
static std::string MY_NAME = "BoGo";
static void My_Function() {}
```

Though we can avoid the external linkage problem by using **static** keyword, it still has issue of polluting global namespace. So, the batter solution is to use **anonymous namespace** as shown below:

```
namespace
{
   const int MY_CONSTANT = 199;
   std::string MY_NAME = "BoGo";
   void My_Function() {}
}
```