

# Being Lazy About Global-Object Initialization

*September 1996*

*by Glenn P. Downing*

## *Introduction*

---

Global data has always been problematic in programming: from Fortran common blocks to C++ static objects. In C++, at least, we can implement global data as a class object which can have a public and non-public interface to provide abstraction and encapsulation. However, even in C++, how we initialize these objects is somewhat problematic.

C++ offers the following implementations of a global object:

- a global or class static object
- a global or class static pointer to a heap object
- a local static object
- a local static pointer to a heap object

In deciding how to implement a global object consider the following issues:

- When will C++ construct and destruct a global object? C++ may construct a global or class static object before the first statement of `main()`, and will destruct it after `main()` returns [2, pp. 19-21]. C++ constructs a local static object the first time it calls the function it's in, and destructs it after `main()` returns [2, pp. 91-92].
- Can we specify the order of initialization among global objects? C++ guarantees the order of initialization of global and class statics within a translation unit, but does not specify an order among different translation units [2, pp. 19-21]. C++ initializes local statics in the order that it calls their respective functions [2, pp. 91-92].
- Will C++ construct and destruct a object, even if it's never used? It will for global and class static objects. It will not for local static objects.
- Are their performance overheads when C++ constructs and destructs a global object or when it's used?

This article presents four approaches, and each provides a different set of trade-offs in terms of these issues.

## *The Global-Static-Object Approach*

---

This approach consists of declaring extern a global static object in the header file and defining that global static object in the source file:

```
// -----  
// Bear.h  
// -----  
  
struct Bear { ... };  
  
// declare a global static object for Pooh
```

```
extern Bear xPooh;

// -----
// Bear.c
// -----

#include "Bear.h"

// define the global static object for Pooh
Bear xPooh(...);
```

Make references through ::xPooh:

```
// -----
// main.c
// -----

#include "Bear.h"

int main () {
    // invoke a method on Pooh
    ::xPooh.vFindHoney();}
```

C++ may construct Pooh before the first statement of main(), and will destruct it after main() returns.

If there were an analogous definition of a global Heffalump, for example, we would not be able to specify an order of initialization between Pooh and the Heffalump, short of defining both of them in the same translation unit.

C++ will construct and destruct Pooh, even if it's never used.

There are no performance overheads when C++ constructs or destructs Pooh, or when it's used.

This approach is both simple and adequate in a large number of cases. However, having C++ construct and destruct Pooh outside the bounds of main() poses some restrictions. In particular, Pooh must not depend on anything that is not also well-defined outside the bounds of main(). For example, we might turn on a memory checker as the first statement of main() and turn it off as the last statement of main(). In that case, the memory checker would not be able to meter the work of the constructor or destructor. Also, some class libraries provide classes that are not well-defined outside the bounds of main().

### *The Class-Static-Pointer-With-Implicit-Initialization Approach*

---

This approach allows us to specify an order of initialization among global objects, even if they are defined in different translation units. It uses reference counting and Scott Meyers describes it completely in his book [1, pp. 178-182].

```
// -----
// Bear.h
// -----

struct Bear {
    // declare a class static pointer for Pooh
    static Bear* pPooh;};
```

```
// -----
// Bear.c
// -----

#include "Bear.h"

// define the class static pointer for Pooh
Bear* Bear::pPooh;
```

A class, `BearInit`, using reference counting, will create and destroy `Pooh`, and store it in `Bear::pPooh`:

```
// -----
// BearInit.h
// -----

#include "Bear.h"

class BearInit {
public:
    BearInit () {
        if (!count++)
            // create Pooh
            Bear::pPooh = new Bear(...);}

    ~BearInit () {
        if (--count)
            // destroy Pooh
            delete Bear::pPooh;}

private:
    // declare a class static object for count
    static int count;};

// define a global static object for BearInit
static BearInit xTheBearInit;

// -----
// BearInit.c
// -----

#include "BearInit.h"

// define the class static object for count
int BearInit::count;
```

Make references through `Bear::pPooh`:

```
// -----
// main.c
// -----

#include "BearInit.h"

int main () {
    // invoke a method on Pooh
    Bear::pPooh->vFindHeffalump();}
```

C++ may construct Pooh before the first statement of `main()`, and will destruct it after `main()` returns.

If there were an analogous definition of a global Heffalump, we would specify the order of initialization by the declared dependency between Pooh and the Heffalump, and we would be specifying that order at compile-time.

C++ will construct and destruct Pooh, even if it's never used.

There is a minimal performance overhead when C++ constructs and destructs Pooh, because of the reference counting and use of the heap. And there is a slight performance overhead when it's used, because of the pointer indirection to the heap.

Again, this approach poses similar restrictions to that of the Global-Static-Approach. Pooh must not depend on anything that is not also well-defined outside the bounds of `main()`.

### *The Class-Static-Pointer-with-Explicit-Initialization Approach*

---

This approach does not construct or destruct Pooh outside the bounds of `main()`. It uses a class static pointer for Pooh and it creates and destroys Pooh as the first and last statements of `main()`:

```
// -----
// Bear.h
// -----

struct Bear {
    // declare a class static pointer for Pooh
    static Bear* pPooh;};

// -----
// Bear.c
// -----

#include "Bear.h"

// define the class static pointer for Pooh
Bear* Bear::pPooh;
```

Make references through `Bear::pPooh`:

```
// -----
// main.c
// -----

#include "Bear.h"

int main () {
    // create Pooh
    Bear::pPooh = new Bear(...);

    // invoke a method on Pooh
    Bear::pPooh->vFindHeffalump();

    // destroy Pooh
    delete Bear::pPooh;}


```

C++ constructs and destructs Pooh within the bounds of main().

If there were an analogous definition of a global Heffalump, we would specify the order of initialization by our explicit order of creation and destruction between Pooh and the Heffalump, and we would be specifying that order at compile-time.

C++ will construct and destruct Pooh, even if it's never used.

There is a minimal performance overhead when C++ constructs and destructs Pooh, because of the heap. And there is a slight performance overhead when it's used, because of the pointer indirection to the heap.

This approach addresses the problem of construction and destruction outside the bounds of main(), but it is a very manual solution, in that we must explicitly create the global objects in the correct order at the beginning of main() and then destroy them in the reverse order at the end of main().

### *The Class-Static-Pointer-with-Lazy-Initialization Approach*

---

The final approach involves a lazy-evaluation strategy, a term coined by functional-programming designers. The idea is to not create Pooh until we need it. We can do this by creating it within a static method:

```
// -----
// Bear.h
// -----

struct Bear {
    // define the static method
    static Bear& rPooh() {
        if (!pPooh) {
            // create Pooh
            pPooh = new Bear(...);}

        // return Pooh
        return *pPooh;}

    // declare a class static pointer for Pooh
    static Bear* pPooh;};

// -----
// Bear.c
// -----

#include "Bear.h"

// define the class static pointer for Pooh
Bear* Bear::pPooh;
```

Make references through Bear::rPooh():

```
// -----
// main.c
// -----

#include "Bear.h"
```

```
int main () {
    // invoke a method on Pooh
    Bear::rPooh().vFindHeffalump();

    // destroy Pooh
    delete Bear::pPooh;}

```

C++ will not construct and destruct Pooh, if it's never used.

The problem is that if there is a global Heffalump, we elegantly specify the constructor order at run time, but we non-elegantly specify the destructor order at compile time.

The solution is to create a class, Registry, in which we will register all the global objects of the program. The first statement of main() will create the Registry, and the last statement will destroy it. That is, we will use the Class-Static-Pointer-with-Explicit-Initialization approach for the Registry.

Staying within the bounds of a strongly-typed solution, how can the Registry call the destructors for various other objects of different types without knowing their types at compile time, which of course would defeat the purpose? It can do this by having every class of a global object define a static method, vDestroy, that in turn deletes that global object and invokes its destructor. The Registry can then have an array of function pointers that point to those vDestroy static methods, and upon termination, invoke those methods in the reverse order of their corresponding construction:

```
// -----
// Registry.h
// -----

class Registry {
public:
    Registry () : iIndex(0) {
        // initialize the function-pointer array
        for (int i = 0; i < iSize; i++)
            aFunctions[ i] = 0;}

    ~Registry () {
        // invoke the vDestroy methods in reverse
        for (int i = iIndex - 1; i &= 0; i--)
            (*aFunctions[ i] ) ();}

    // declare a class static pointer for the Registry
    static Registry* pTheRegistry;

private:
    // define a typedef for the vDestroy methods
    typedef void (*FP) ();

public:
    // define a method to store the function pointers
    static void vStore (FP pFunction) {
        assert(iIndex < eSize);
        aFunctions[ iIndex++] = pFunction;}

private:
    // define the size of the array
    enum {eSize = 16};
};

```

```

        // define an index into the array
        int iIndex;

        // define the function-pointer array
        FP aFunctions[ eSize] ;};

// -----
// Registry.c
// -----

#include "Registry.h"

// define the class static pointer for the Registry
Registry* Registry::pTheRegistry;

```

Now we're almost done. We just need to revise the definition of Bear slightly:

```

// -----
// Bear.h
// -----

class Bear {
public:
    static Bear& rPooh () {
        if (!pPooh) {
            // create Pooh
            pPooh = new Bear(...);

            // store a pointer to vDestroy
            Registry::vStore (&Bear::vDestroy);

            // return Pooh
            return *pPooh;}

    // declare a class static pointer for Pooh
    static Bear* pPooh;

private:
    static void vDestroy () {
        // destroy Pooh
        delete pPooh;}};

```

Make references through Bear::rPooh():

```

// -----
// main.c
// -----

#include "Bear.h"

int main () {
    // create the Registry
    Registry::pTheRegistry = new Registry;

    // invoke a method on Pooh
    Bear::rPooh().vFindHeffalump();
}

```

```
// destroy the Registry
delete Registry::pTheRegistry;}
```

C++ constructs and destructs Pooh within the bounds of main().

If there were an analogous definition of a global Heffalump, we would specify the order of initialization by our order of use of Pooh and the Heffalump, and we would be specifying that order at run-time!

C++ will not construct and destruct Pooh, if it's never used.

There is a performance overhead when C++ constructs and destructs Pooh, because of the conditional to check if Pooh has already been created and because of the use of the heap. And there is a slight performance overhead when Pooh is used, because of the pointer indirection to the heap.

### *Discussion*

---

The Global-Static-Object approach is the simplest, and within its limitations, a very nice solution.

The Class-Static-Pointer-with-Implicit-Initialization approach is more complex, but necessary if we need to define global objects outside the bounds of main().

I don't recommend the Class-Static-Pointer-with-Explicit-Initialization approach. It's too manual and therefore error prone.

I recommend the Class-Static-Pointer-with-Lazy-Initialization approach. The approach is very dynamic. The vDestroy methods create the global objects in the order dictated by their interdependencies and we can specify that order at run time. As well, the global object is never created if it isn't used. And finally, the Registry destroys the global objects in the reverse order of their construction.

### *References*

---

1. Meyers, S. Effective C++, Reading, MA, Addison-Wesley, 1992.
2. Stroustrup, B. and M. Ellis. The Annotated C++ Reference Manual, Reading, MA, Addison-Wesley, 1990.

*Copyright ' 1996 Journal of Object-Oriented Programming (JOOP).*  
*All rights reserved.*