6.7 — External linkage

BY ALEX ON JANUARY 3RD, 2020 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 7TH, 2020

In the prior lesson (**6.6 -- Internal linkage**), we discussed how `internal linkage` limits the use of an identifier to a single file. In this lesson, we'll explore the concept of `external linkage`.

An identifier with **external linkage** can be seen and used both from the file in which it is defined, and from other code files (via a forward declaration). In this sense, identifiers with external linkage are truly "global" in that they can be used anywhere in your program!

Functions have external linkage by default

In lesson **2.8 -- Programs with multiple code files**, you learned that you can call a function defined in one file from another file. This is because functions have external linkage by default.

In order to call a function defined in another file, you must place a `forward declaration` for the function in any other files wishing to use the function. The forward declaration tells the compiler about the existence of the function, and the linker connects the function calls to the actual function definition.

Here's an example:

a.cpp:

```
#include <iostream>

void sayHi() // this function has external linkage, and can be seen by other files
{
    std::cout << "Hi!";
}
```

main.cpp:

```
void sayHi(); // forward declaration for function sayHi, makes sayHi accessible in this file

int main()
{
    sayHi(); // call to function defined in another file, linker will connect this call to the function definition

    return 0;
}
```

The above program prints:

```
Hi!
```

In the above example, the forward declaration of function `sayHi()` in `main.cpp` allows `main.cpp` to access the `sayHi()` function defined in `a.cpp`. The forward declaration satisfies the compiler, and the linker is able to link the function call to the function definition.

If function `sayHi()` had internal linkage instead, the linker would not be able to connect the function call to the function definition, and a linker error would result.

Global variables with external linkage

Global variables with external linkage are sometimes called **external variables**. To make a global variable external (and thus accessible by other files), we can use the `extern` keyword to do so:

```
int g_x { 2 }; // non-constant globals are external by default

extern const int g_y { 3 }; // const globals can defined as extern, making them external
extern constexpr int g_z { 3 }; // constexpr globals can defined as extern, making them external (but this is useless, see the note in the next section)

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

Non-const global variables are external by default (if used, the `extern` keyword will be ignored).

Variable forward declarations via the extern keyword

To actually use an external global variable that has been defined in another file, you also must place a `forward declaration` for the global variable in any other files wishing to use the variable. For variables, creating a forward declaration is also done via the `extern` keyword (with no initialization value).

Here is an example of using a variable forward declaration:

a.cpp:

```
// global variable definitions
int g_x { 2 }; // non-constant globals have external linkage by default
extern const int g_y { 3 }; // this extern gives g_y external linkage
```

main.cpp:

```
#include <iostream>

extern int g_x; // this extern is a forward declaration of a variable named g_x that is defined somewhere else
extern const int g_y; // this extern is a forward declaration of a const variable named g_y that is defined somewhere else

int main()
{
    std::cout << g_x; // prints 2

    return 0;
}
```

In the above example, `a.cpp` and `main.cpp` both reference the same global variable named g_x. So even though g_x is defined and initialized in `a.cpp`, we are able to use its value in `main.cpp` via the forward declaration of g_x.

Note that the `extern` keyword has different meanings in different contexts. In some contexts, `extern` means "give this variable external linkage". In other contexts, `extern` means "this is a forward declaration for an external variable that is defined somewhere else". Yes, this is confusing, so we summarize all of these usages in lesson **6.11 -- Scope, duration, and linkage summary**.

Warning

If you want to define an uninitialized non-const global variable, do not use the extern keyword, otherwise C++ will think you're trying to make a forward declaration for the variable.

Warning

Although constexpr variables can be given external linkage via the `extern` keyword, they can not be forward declared, so there is no value in giving them external linkage.

Note that function forward declarations don't need the `extern` keyword -- the compiler is able to tell whether you're defining a new function or making a forward declaration based on whether you supply a function body or not. Variables forward declarations *do* need the `extern` keyword to help differentiate variables definitions from variable forward declarations (they look otherwise identical):

```
// non-constant
int g_x; // variable definition (can have initializer if desired)
extern int g_x; // forward declaration (no initializer)

// constant
extern const int g_y { 1 }; // variable definition (const requires initializers)
extern const int g_y; // forward declaration (no initializer)
```

File scope vs. global scope

The terms "file scope" and "global scope" tend to cause confusion, and this is partly due to the way they are informally used. Technically, in C++, *all* global variables in C++ have "file scope", and the linkage property controls whether they can be used in other files or not.

Consider the following program:

global.cpp:

```
int g_x { 2 }; // external linkage by default
// g_x goes out of scope here
```

main.cpp:

```
extern int g_x; // forward declaration for g_x -- g_x can be used beyond this point in this fi
le

int main()
{
    std::cout << g_x; // should print 2

    return 0;
}
// the forward declaration for g_x goes out of scope here
```

Variable g_x has file scope within `global.cpp` -- it can be used from the point of definition to the end of the file, but it can not be directly seen outside of `global.cpp`.

Inside `main.cpp`, the forward declaration of g_x also has file scope -- it can be used from the point of declaration to the end of the file.

However, informally, the term "file scope" is more often applied to global variables with internal linkage, and "global scope" to global variables with external linkage (since they can be used across the whole program, with the appropriate forward declarations).

The initialization order problem of global variables

Initialization of global variables happens as part of program startup, before execution of the `main` function. This proceeds in two phases.

The first phase is called `static initialization`. In the static initialization phase, global variables with constexpr initializers (including literals) are initialized to those values. Also, global variables without initializers are zero-initialized.

The second phase is called `dynamic initialization`. This phase is is more complex and nuanced, but the gist of it is that global variables with non-constexpr initializers are initialized.

Here's an example of a non-constexpr initializer:

```
int init()
{
    return 5;
}

int g_something{ init() }; // non-constexpr initialization
```

Within a single file, global variables are generally initialized in order of definition (there are a few exceptions to this rule). Given this, you need to be careful not to have variables dependent on the initialization value of other variables that won't be initialized until later. For example:

```
#include <iostream>

int initx();  // forward declaration
int inity();  // forward declaration

int g_x{ initx() }; // g_x is initialized first
int g_y{ inity() };

int initx()
{
    return g_y; // g_y isn't initialized when this is called
}

int inity()
{
  return 5;
}

int main()
{
    std::cout << g_x << ' ' << g_y << '\n';
}
```

This prints:

```
0 5
```

Much more of a problem, the order of initialization across different files is not defined. Given two files, `a.cpp` and `b.cpp`, either could have its global variables initialized first. This means that if the variables in `a.cpp` are dependent upon the values in `b.cpp`, there's a 50% chance that those variables won't be initialized yet.

Warning

Dynamic initialization of global variables causes a lot of problems in C++. Avoid it whenever possible.

Quick summary

```
// External global variable definitions:
int g_x;                       // defines non-initialized external global variable (zero initi
alized by default)
extern const int g_x{ 1 };     // defines initialized const external global variable
extern constexpr int g_x{ 2 }; // defines initialized constexpr external global variable (

// Forward declarations
extern int g_y;                // forward declaration for non-constant global variable
extern const int g_y;          // forward declaration for const global variable
extern constexpr int g_y;      // not allowed: constexpr variables can't be forward declared
```

We provide a comprehensive summary in lesson **6.11 -- Scope, duration, and linkage summary**.

Quiz time

Question #1

What's the difference between a variable's scope, duration, and linkage? What kind of scope, duration, and linkage do global variables have?

Show Solution

6.8 -- Global constants and inline variables

Index

6.6 -- Internal linkage

C++ TUTORIAL | PRINT THIS POST

11 comments to 6.7 — External linkage

luisizq
February 6, 2020 at 3:12 pm · Reply

In the section "Variable forward declarations via the extern keyword" I think there is an error in an example

Instead of:

// non-constant
int g_x; // variable definition (can have initializer if desired)
extern int g_x; // forward declaration (no initializer)

// constant
extern const g_y { 1 }; // variable definition (const requires initializers)
extern const g_y; // forward declaration (no initializer)

There should be

// non-constant
int g_x; // variable definition (can have initializer if desired)
extern int g_x; // forward declaration (no initializer)

// constant
extern const int g_y { 1 }; // variable definition (const requires initializers)
extern const int g_y; // forward declaration (no initializer)

nascardriver
February 7, 2020 at 8:47 am · Reply

Yep, thanks!

Jojo
January 25, 2020 at 6:05 am · Reply

Hello,
I am confused by title "The static initialization order problem": While reading the following content, I thought more about a title like "The problem while initializing static variables". Indeed, the problem concerns the dynamic initialization of static variables rather than their static initialization?

nascardriver
January 25, 2020 at 6:29 am · Reply

Good catch!
static initialization isn't the problem, dynamic initialization is. Title updated to "The initialization order problem of global variables".

Tim
January 20, 2020 at 10:41 am · Reply

Been learning cpp now for around 4 years, mostly thanks to this site I think I am reasonably competent now, but I still find it incredibly useful reference material for stuff that I do not do very often.

I have started using a global stream for my logs as passing the object to every single class just seemed needlessly messy and the built in streams could not be used because I a writing a daemon, but I was getting linker errors as I had not forward declared my variable where I initialised it in main. I still have it declared extern in my header file though, it seems to work, but is that wrong?

nascardriver
January 21, 2020 at 2:22 am · Reply

Declare it `extern` in the header, define it once in a single source fail.

Tim
January 21, 2020 at 4:53 pm · Reply

Thanks for the confirmation.

Armitage
January 8, 2020 at 12:03 pm · Reply

I'm confused about "static variables" term in section "The static initialization order problem".

```
int g_x = initx()
```

g_x— is a static variable?

Alex
January 8, 2020 at 6:16 pm · Reply

In this context, "static variable" means a variable with static duration. Since all globals have static duration, g_x is indeed a static variable.

Thomas
January 7, 2020 at 12:11 pm · Reply

At the beginning of this lesson, it says : "In the prior lesson (8.5b -- Non-static member initialization)". Just to let you know, if you want to put the correct one (6.6 — Internal linkage).

nascardriver
January 8, 2020 at 3:55 am · Reply

Link updated, thanks!