

## 6.7 — External linkage

ALEX JUNE 17, 2021

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 [2.7 -- 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:

```
1 | #include <iostream>
2 | void sayHi() // this function has external linkage, and can be seen by other
3 | files
   | {
   |     std::cout << "Hi!";
   | }
```

main.cpp:

```

1 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:

```

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

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

int main()
{
    return 0;
2 }

```

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`:

```

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

```

`main.cpp`:

```

1 #include <iostream>

2 extern int g_x; // this extern is a forward declaration of a variable named g_x that is defined
3 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):

```
1 // non-constant
  int g_x; // variable definition (can have initializer if desired)
2 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 have “file scope”, and the linkage property controls whether they can be used in other files or not.

Consider the following program:

global.cpp:

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

main.cpp:

```
1 extern int g_x; // forward declaration for g_x -- g_x can be used beyond this point in this
  file
  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).

---

## Quick summary

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



### Next lesson

**6.8** Why (non-const) global variables are evil



[Back to table of contents](#)



### Previous lesson

**6.6** Internal linkage


Leave a comment... Put C++ code between triple-backticks (markdown style): ``Your C++ code``

Name\*

Email\*

?

Avatars from <https://gravatar.com/> are connected to your provided email address.

Notify me about replies: 

POST COMMENT

DP N N FOUT

Newest ▼

