4.2 — Global variables and linkage

BY ALEX ON JUNE 19TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 13TH, 2018

In the last lesson, you learned that variables declared inside a function are called local variables. Local variables have block scope (they are only visible within the block they are declared), and have automatic duration (they are created at the point of definition and destroyed when the block is exited).

Variables declared *outside* of a function are called **global variables**. Global variables have **static duration**, which means they are created when the program starts and are destroyed when it ends. Global variables have **file scope** (also informally called "global scope" or "global namespace scope"), which means they are visible until the end of the file in which they are declared.

Defining global variables

By convention, global variables are declared at the top of a file, below the includes, but above any code. Here's an example of a couple of global variables being defined.

```
#include <iostream>

// Variables declared outside of a function are global variables
int g_x; // global variable g_x
const int g_y(2); // global variable g_y

void doSomething()
{
    // global variables can be seen and used everywhere in program
    g_x = 3;
    std::cout << g_y << "\n";
}

int main()
{
    doSomething();

    // global variables can be seen and used everywhere in program
    g_x = 5;
    std::cout << g_y << "\n";

    return 0;
}
```

Similar to how variables in an inner block with the same name as a variable in an outer block hides the variable in the outer block, local variables with the same name as a global variable hide the global variable inside the block that the local variable is declared in. However, the global scope operator (::) can be used to tell the compiler you mean the global version instead of the local version.

```
#include <iostream>
int value(5); // global variable

int main()
{
    int value = 7; // hides the global variable value
    value++; // increments local value, not global value
    ::value--; // decrements global value, not local value

    std::cout << "global value: " << ::value << "\n";
    std::cout << "local value: " << value << "\n";
    return 0;
} // local value is destroyed
```

This code prints:

```
global value: 4
local value: 8
```

However, having local variables with the same name as global variables is usually a recipe for trouble, and should be avoided whenever possible. By convention, many developers prefix global variable names with “g_” to indicate that they are global. This both helps identify global variables as well as avoids naming conflicts with local variables.

Internal and external linkage via the static and extern keywords

In addition to scope and duration, variables have a third property: linkage. A variable’s **linkage** determines whether multiple instances of an identifier refer to the same variable or not.

A variable with no linkage can only be referred to from the limited scope it exists in. Normal local variables are an example of variables with no linkage. Two local variables with the same name but defined in different functions have no linkage -- each will be considered an independent variable.

A variable with internal linkage is called an **internal variable** (or static variable). Variables with internal linkage can be used anywhere within the file they are defined in, but can not be referenced outside the file they exist in.

A variable with external linkage is called an **external variable**. Variables with external linkage can be used both in the file they are defined in, as well as in other files.

If we want to make a global variable internal (able to be used only within a single file), we can use the **static** keyword to do so:

```
static int g_x; // g_x is static, and can only be used within this file

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

Similarly, if we want to make a global variable external (able to be used anywhere in our program), we can use the **extern** keyword to do so:

```
extern double g_y(9.8); // g_y is external, and can be used by other files

// Note: those other files will need to use a forward declaration to access this external variable
// We'll discuss this in the next section

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

By default, non-const variables declared outside of a function are assumed to be external. However, const variables declared outside of a function are assumed to be internal.

Variable forward declarations via the extern keyword

In the section on **programs with multiple files**, you learned that in order to use a function declared in another file, you have to use a function forward declaration.

Similarly, in order to use an external global variable that has been declared in another file, you *must* use a variable forward declaration. For variables, creating a forward declaration is done via the **extern** keyword (with no initialization value).

Note that this means 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”. We’ll summarize these usages in lesson **4.3a -- Scope, duration, and linkage summary**.

Here is an example of using a variable forward declaration:

global.cpp:

```
// define two global variables
// non-const globals have external linkage by default
int g_x; // external linkage by default
extern int g_y(2); // external linkage by default, so this extern is redundant and ignored

```

```
// in this file, g_x and g_y can be used anywhere beyond this point
```

main.cpp:

```
extern int g_x; // forward declaration for g_x (defined in global.cpp) -- g_x can now be used beyond this point in this file

int main()
{
    extern int g_y; // forward declaration for g_y (defined in global.cpp) -- g_y can be used beyond this point in main() only

    g_x = 5;
    std::cout << g_y; // should print 2

    return 0;
}
```

If the variable forward declaration is declared outside of a function, it applies for the whole file. If the variable forward declaration is declared inside a function, it applies within that block only.

If a variable is declared as static, trying to use a forward declaration to access it will not work:

constants.cpp:

```
static const double g_gravity(9.8);
```

main.cpp:

```
#include <iostream>

extern const double g_gravity; // This will satisfy the compiler that g_gravity exists

int main()
{
    std:: cout << g_gravity; // This will cause a linker error because the only definition of g_gravity is inaccessible from here
    return 0;
}
```

Note that 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.

Function linkage

Functions have the same linkage property that variables do. Functions always default to external linkage, but can be set to internal linkage via the static keyword:

```
// This function is declared as static, and can now be used only within this file
// Attempts to access it via a function prototype will fail
static int add(int x, int y)
{
    return x + y;
}
```

Function forward declarations don't need the extern keyword. The compiler is able to tell whether you're defining a function or a function prototype by whether you supply a function body or not.

The one-definition rule and non-external linkage

In lesson **1.7 -- Forward declarations and definitions**, we noted that the one-definition rule says that an object or function can't have more than one definition, either within a file or a program.

However, it's worth noting that non-extern objects and functions in different files are considered to be different entities, even if their names and types are identical. This makes sense, since they can't be seen outside of their respective files anyway.

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”. However, informally, the term “file scope” is more often applied to file scope variables with internal linkage only, and “global scope” to file scope variables with external linkage.

Consider the following program:

global.cpp:

```
int g_x(2); // external linkage by default
```

main.cpp:

```
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;
}
```

g_x has file scope within global.cpp -- it can not be directly seen outside of global.cpp. Note that even though it’s used in main.cpp, main.cpp isn’t seeing g_x, it’s seeing the forward declaration of g_x (which also has file scope). The linker is responsible for linking up the definition of g_x in global.cpp with the use of g_x in main.cpp.

Global symbolic constants

In section **2.9 -- Symbolic constants and the const keyword**, we introduced the concept of symbolic constants, and defined them like this:

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

// define your own namespace to hold constants
namespace Constants
{
    const double pi(3.14159);
    const double avogadro(6.0221413e23);
    const double my_gravity(9.2); // m/s^2 -- gravity is light on this planet
    // ... other related constants
}
#endif
```

While this is simple (and fine for smaller programs), every time constants.h gets #included into a different code file, each of these variables is copied into the including code file. Therefore, if constants.h gets included into 20 different code files, each of these variables is duplicated 20 times. Header guards won’t stop this from happening, as they only prevent a header from being included more than once into a single including file, not from being included one time into multiple different code files. This duplication of variables isn’t really that much of a problem (since constants aren’t likely to be huge), but changing a single constant value would require recompiling every file that includes the constants header, which can head to lengthy rebuild times for larger projects.

We can avoid this problem by turning these constants into const global variables, and changing the header file to hold only the variable forward declarations:

constants.cpp:

```
namespace Constants
{
    // actual global variables
    extern const double pi(3.14159);
    extern const double avogadro(6.0221413e23);
    extern const double my_gravity(9.2); // m/s^2 -- gravity is light on this planet
}
```

constants.h:

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

namespace Constants
{
    // forward declarations only
    extern const double pi;
    extern const double avogadro;
    extern const double my_gravity;
}

#endif
```

Use in the code file stays the same:

```
#include "constants.h"
double circumference = 2 * radius * Constants::pi;
```

Because global symbolic constants should be namespaced (to avoid naming conflicts with other identifiers in the global namespace) and are read-only, the use of the g_ prefix is not necessary.

Now the symbolic constants will get instantiated only once (in constants.cpp), instead of once every time constants.h is #included, and the other uses will simply refer to the version in constants.cpp. Any changes made to constants.cpp will require recompiling only constants.cpp.

However, there are a couple of downsides to doing this. First, these constants are now considered compile-time constants only within the file they are actually defined in (constants.cpp), not anywhere else they are used. This means that outside of constants.cpp, they can't be used anywhere that requires a compile-time constant (such as for the length of a fixed array, something we talk about in chapter 6). Second, the compiler may not be able to optimize these as much.

Given the above downsides, we recommend defining your constants in the header file. If you find that for some reason those constants are causing trouble, you can move them into a .cpp file as per the above as needed.

A word of caution about (non-const) global variables

New programmers are often tempted to use lots of global variables, because they are easy to work with, especially when many functions are involved. However, use of non-const global variables should generally be avoided altogether! We'll discuss why in the next section.

Summary

Global variables have global scope, and can be used anywhere in the program. Like functions, you must use a forward declaration (via keyword extern) to use a global variable defined in another file.

By default, non-const global variables have external linkage. You can use the static keyword to explicitly make them internal if desired. By default, const global variables have internal linkage. You can use the extern keyword to explicitly make them external if desired.

Use a g_ prefix to help identify your non-const global variables.

Here's a summary chart of the use of the extern and static keywords for non-const and const variable use cases:

```
// Uninitialized definition:
int g_x;        // defines uninitialized global variable (external linkage)
static int g_x; // defines uninitialized static variable (internal linkage)
const int g_x;  // not allowed: const variables must be initialized

// Forward declaration via extern keyword:
extern int g_z;       // forward declaration for global variable defined elsewhere
extern const int g_z; // forward declaration for const global variable defined elsewhere

// Initialized definition:
int g_y(1);        // defines initialized global variable (external linkage)
static int g_y(1); // defines initialized static variable (internal linkage)
const int g_y(1);  // defines initialized static variable (internal linkage)

// Initialized definition w/extern keyword:
```

```
extern int g_w(1);       // defines initialized global variable (external linkage, extern keyword is redundant in this case)
extern const int g_w(1); // defines initialized const global variable (external linkage)
```

Quiz

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

Quiz Solutions

1) **Hide Solution**

Scope determines where a variable is accessible. Duration determines where a variable is created and destroyed. Linkage determines whether the variable can be exported to another file or not.

Global variables have global scope (aka. file scope), which means they can be accessed from the point of declaration to the end of the file in which they are declared.

Global variables have static duration, which means they are created when the program is started, and destroyed when it ends.

Global variables can have either internal or external linkage, via the static and extern keywords respectively.

4.2a -- Why global variables are evil

Index

4.1a -- Local variables, scope, and duration

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

188 comments to 4.2 — Global variables and linkage

« Older Comments 1 2 3

Marcos
June 23, 2018 at 8:52 am · Reply

In the above example for 'Global Symbolic Constants' why does both constants.h and constants.cpp use the namespace 'Constants'? I removed the namespace in the header and adjusted the reference but it called a linker error, presumably because the forward declarations didn't describe the namespace used in the initialization? I then tried removing the namespace from the constants.cpp and keeping the namespace in the header file and reference but it again provided the same linker error.

Alex
June 29, 2018 at 2:14 pm · Reply

The compiler needs to know that the forward declarations in the header are inside the namespace so it can enforce the proper syntax, and the linker needs to know that the definitions inside the code file are also inside