6.6 — Internal linkage

BY ALEX ON JANUARY 3RD, 2020 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 4TH, 2020

In lesson **6.3 -- Local variables**, we said, "An identifier's linkage determines whether other declarations of that name refer to the same object or not", and we discussed how local variables have no `linkage`.

Global variable and functions identifiers can have either `internal linkage` or `external linkage`. We'll cover the internal linkage case in this lesson, and the external linkage case in lesson **6.7 -- External linkage**.

An identifier with **internal linkage** can be seen and used within a single file, but it is not accessible from other files (that is, it is not exposed to the linker). This means that if two files have identically named identifiers with internal linkage, those identifiers will be treated as independent.

Global variables with internal linkage

Global variables with internal linkage are sometimes called **internal variables**.

To make a non-constant global variable internal, we use the `static` keyword.

```cpp
static int g_x; // non-constant globals have external linkage by default, but can be given internal linkage via the static keyword

const int g_y { 1 }; // const globals have internal linkage by default
constexpr int g_z { 2 }; // constexpr globals have internal linkage by default

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

Const and constexpr global variables have internal linkage by default (and thus don't need the `static` keyword -- if it is used, it will be ignored).

Here's an example of multiple files using internal variables:

a.cpp:

```cpp
constexpr int g_x { 2 }; // this internal g_x is only accessible within a.cpp
```

main.cpp:

```cpp
#include <iostream>

static int g_x { 3 }; // this separate internal g_x is only accessible within main.cpp

int main()
{
    std::cout << g_x << '\n'; // uses main.cpp's g_x, prints 3

    return 0;
}
```

This program prints:

```
3
```

Because g_x is internal to each file, `main.cpp` has no idea that `a.cpp` also has a variable named g_x (and vice versa).

For advanced readers

The use of the `static` keyword above is an example of a **storage class specifier**, which sets both the name's linkage and its storage duration (but not its scope). The most commonly used `storage class specifier` values are `static`, `extern`, and `mutable`. The term `storage class specifier` is mostly used in technical documentations.

The one-definition rule and internal linkage

In lesson **2.7 -- 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 internal objects (and functions) that are defined in different files are considered to be independent entities (even if their names and types are identical), so there is no violation of the one-definition rule. Each internal object only has one definition.

Functions with internal linkage

Because linkage is a property of an identifier (not of a variable), functions have the same linkage property that variables do. Functions default to external linkage (which we'll cover in the next lesson), but can be set to internal linkage via the `static` keyword:

add.cpp:

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

main.cpp:

```
#include <iostream>

int add(int x, int y); // forward declaration for function add

int main()
{
    std::cout << add(3, 4) << '\n';

    return 0;
}
```

This program won't link, because function add is not accessible outside of `add.cpp`.

Quick Summary

```
// Internal global variables definitions:
static int g_x;          // defines non-initialized internal global variable (zero initialized
by default)
static int g_x{ 1 };     // defines initialized internal global variable
```

```

const int g_y { 2 };      // defines initialized internal global const variable
constexpr int g_y { 3 }; // defines initialized internal global constexpr variable

// Internal function definitions:
static int foo() {};      // defines internal function
```

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

6.7 -- External linkage

Index

6.5 -- Variable shadowing (name hiding)

C++ TUTORIAL | PRINT THIS POST

6 comments to 6.6 — Internal linkage

Tepy
February 9, 2020 at 4:08 pm · Reply

"Because linkage is a property of an identifier (not of a variable), functions have the same linkage property that variables do. "

Should this be rephrased to:
"Because linkage is a property of an identifier (not of a variable), functions have the same linkage property that identifiers do."

Abraxasknister
February 5, 2020 at 1:53 pm · Reply

How does this interact with #include? For example if I have

```
// constants.hpp
#pragma once
constexpr double g{ 9.8 };

// velocity.cpp
#include "constants.hpp"
double mps(double sec)
{ return sec*g; }

// acceleration.cpp
#include "constants.hpp"
double mpss(double sec)
{ return sec*sec*g/2.0; }

// main.cpp
```

```
double mps(double);
double mpss(double);

int main(){ /* some code containing mps or mpss */ return 0; }
```

would then mps and mpss each utilize their own ever lasting copy of g so that the purpose of having that g centralised is defeated because it's stored twice? (I'm guessing that the better solution would be to have it in a constants namespace).

nascardriver
February 6, 2020 at 9:21 am · Reply

Each get their own `g`. Making `g` `inline` solves this.

```
constexpr inline g{ 9.8 };
```

kavin
January 10, 2020 at 7:32 am · Reply

I was thinking that till now that warnings are actually legit errors in main code that program would ignore to compile if "Treat warnings as errors" was disabled. Thanks for clearing my doubt @nascardriver

kavin
January 7, 2020 at 6:49 am · Reply

Under "Functions with internal linkage", if i add
int add(int x, int y)
{
  return x - y;
}
to main.cpp and remove int add(int x, int y); then i get this error.

Source.cpp(1,12): error C2220: the following warning is treated as an error
Source.cpp(1,12): warning C4505: 'add': unreferenced local function has been removed

why is that? since add.cpp has a static function, the program should auto ignore the function in add.cpp and compile without any problem right ? But it behaves differently here.

nascardriver
January 9, 2020 at 3:03 am · Reply

You're not getting an error as a violation of a language rule.
You're getting an error because you got a warning and your treating warnings as errors.
You're getting a warning as a friendly reminder from the compiler that you wrote a function but you're never using it (`add` in "add.cpp").

The code is fine, but you made a mistake, and you told your compiler not to let you make mistakes. If you want to compile the code anyway, you can temporarily disable "Treat warnings as errors" (or similar) in your project settings.