7.6 — Internal linkage

learncpp.com/cpp-tutorial/internal-linkage/

In lesson 7.3 -- 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 7.7 -- External linkage and variable forward declarations.

An identifier with **internal linkage** can be seen and used within a single translation unit, but it is not accessible from other translation units (that is, it is not exposed to the linker). This means that if two source files have identically named identifiers with internal linkage, those identifiers will be treated as independent (and do not result in an ODR violation for having duplicate definitions).

Related content

We cover translation units in lesson 2.10 -- Introduction to the preprocessor.

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.

```
#include <iostream>

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()
{
    std::cout << g_x << ' ' << g_y << ' ' << g_z << '\n';
    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:

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

main.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. The most commonly used `storage class specifiers` are `static`, `extern`, and `mutable`. The term `storage class specifier` is mostly used in technical documentations.

Functions with internal linkage

Because linkage is a property of an identifier (not of a variable), function identifiers have the same linkage property that variable identifiers 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
[[maybe_unused]] static int add(int x, int y)
{
    return x + y;
}
```

main.cpp:

```
#include <iostream>

static 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`.

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.

`static` vs unnamed namespaces

In modern C++, use of the `static` keyword for giving identifiers internal linkage is falling out of favor. Unnamed namespaces can give internal linkage to a wider range of identifiers (e.g. type identifiers), and they are better suited for giving many identifiers internal linkage.

We cover unnamed namespaces in lesson 7.13 -- Unnamed and inline namespaces.

Why bother giving identifiers internal linkage?

There are typically two reasons to give identifiers internal linkage:

- There is an identifier we want to make sure isn't accessible to other files. This could be a global variable we don't want messed with, or a helper function we don't want called.
- To be pedantic about avoiding naming collisions. Because identifiers with internal linkage aren't exposed to the linker, they can only collide with names in the same translation unit, not across the entire program.

Many modern development guides recommend giving every variable and function that isn't meant to be used from another file internal linkage. If you have the discipline, this is a good recommendation.

For now, we'll recommend a lighter-touch approach as a minimum: give internal linkage to any identifier that you have an explicit reason to disallow access from other files.

Best practice

Give identifiers internal linkage when you have an explicit reason to disallow access from other files.

Consider giving all identifiers you don't want accessible to other files internal linkage (use an unnamed namespace for this).

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 7.11 -- Scope, duration, and linkage summary.