6.11 — Scope, duration, and linkage summary

BY ALEX ON APRIL 19TH, 2016 | LAST MODIFIED BY ALEX ON JANUARY 3RD, 2020

The concepts of scope, duration, and linkage cause a lot of confusion, so we're going to take an extra lesson to summarize everything. Some of these things we haven't covered yet, and they're here just for completeness / reference later.

Scope summary

An identifier's *scope* determines where the identifier can be accessed within the source code.

- Variables with **block scope / local scope** can only be accessed within the block in which they are declared (including nested blocks). This includes:
  - Local variables
  - Function parameters
  - User-defined type definitions (such as enums and classes) declared inside a block
- Variables and functions with **global scope / file scope** can be accessed anywhere in the file. This includes:
  - Global variables
  - Functions
  - User-defined type definitions (such as enums and classes) declared inside a namespace or in the global scope

Duration summary

A variable's *duration* determines when it is created and destroyed.

- Variables with **automatic duration** are created at the point of definition, and destroyed when the block they are part of is exited. This includes:
  - Local variables
  - Function parameters
- Variables with **static duration** are created when the program begins and destroyed when the program ends. This includes:
  - Global variables
  - Static local variables
- Variables with **dynamic duration** are created and destroyed by programmer request. This includes:
  - Dynamically allocated variables

Linkage summary

An identifier's *linkage* determines whether multiple instances of an identifier refer to the same identifier or not.

- An identifier with **no linkage** means the identifier only refers to itself. This includes:
  - Local variables
  - User-defined type definitions (such as enums and classes) declared inside a block
- An identifier with **internal linkage** can be accessed anywhere within the file it is declared. This includes:
  - Static global variables (initialized or uninitialized)
  - Static functions
  - Const global variables
  - Functions declared inside an unnamed namespace
  - User-defined type definitions (such as enums and classes) declared inside an unnamed namespace

- An identifiers with **external linkage** can be accessed anywhere within the file it is declared, or other files (via a forward declaration). This includes:
    - Functions
    - Non-const global variables (initialized or uninitialized)
    - Extern const global variables
    - Inline const global variables
    - User-defined type definitions (such as enums and and classes) declared inside a namespace or in the global scope

Identifiers with external linkage will generally cause a duplicate definition linker error if the definitions are compiled into more than one .cpp file (due to violating the one-definition rule). There are some exceptions to this rule (for types, templates, and inline functions and variables) -- we'll cover these further in future lessons when we talk about those topics.

Also note that functions have external linkage by default. They can be made internal by using the static keyword.

Variable scope, duration, and linkage summary

Because variables have scope, duration, and linkage, let's summarize in a chart:

Type	Example	Scope	Duration	Linkage	Notes
Local variable	int x;	Block	Automatic	None	
Static local variable	static int s_x;	Block	Static	None	
Dynamic variable	int *x { new int };	Block	Dynamic	None	
Function parameter	void foo(int x)	Block	Automatic	None	
External non-constant global variable	int g_x;	File	Static	External	Initialized or uninitialized
Internal non-constant global variable	static int g_x;	File	Static	Internal	Initialized or uninitialized
Internal constant global variable	constexpr int g_x { 1 };	File	Static	Internal	Const or constexpr, Must be initialized
External constant global variable	extern constexpr int g_x { 1 };	File	Static	External	Const or constexpr, Must be initialized
Inline constant global variable	inline constexpr int g_x { 1 };	File	Static	External	Const or constexpr, Must be initialized

Forward declaration summary

You can use a forward declaration to access a function or variable in another file:

Type	Example	Notes
Function forward declaration	void foo(int x);	Prototype only, no function body
Non-constant global variable forward declaration	extern int g_x;	Must be uninitialized
Const global variable forward declaration	extern const int g_x;	Must be uninitialized
Constexpr global variable forward declaration	extern constexpr int g_x;	Not allowed, constexpr cannot be forward declared

What the heck is a storage class specifier?

When used as part of an identifier declaration, the `static` and `extern` keywords are called **storage class specifiers**. In this context, they set the storage duration and linkage of the identifier.

C++ supports 4 active storage class specifiers:

Specifier	Meaning	Note
extern	static (or thread_local) storage duration and external linkage	
static	static (or thread_local) storage duration and internal linkage	
thread_local	thread storage duration	Introduced in C++11
mutable	object allowed to be modified even if containing class is const	
auto	automatic storage duration	Deprecated in C++11
register	automatic storage duration and hint to the compiler to place in a register	Deprecated in C++17

The term *storage class specifier* is typically only used in formal documentation.

6.12 -- Using statements

Index

6.10 -- Static local variables

C++ TUTORIAL | PRINT THIS POST

80 comments to 6.11 — Scope, duration, and linkage summary

« Older Comments 1 2

Anastasia
August 29, 2019 at 5:28 am · Reply

Hi,
I may be dumb, but in the linkage summary it's said that static functions are covered in chapter 7, but I can't see where it is and don't remember learning about them (except for member static functions, which I suppose are a totally different thing and they are covered later). Can anyone point me in the right direction?

Alex
September 1, 2019 at 2:19 pm · Reply

Hmmm, I think I actually don't cover these explicitly. I've removed the reference to chapter 7.

But just like static global variables, static functions are only accessible in the file in which they are declared.

Anastasia
September 1, 2019 at 2:41 pm · Reply

There's a brief explanation (with an example) of static functions in lesson S.4.2 — Global variables and linkage (paragraph 'Function linkage'), it could be used as a reference instead (as a little refresher).

Thanks!

mmp52
August 27, 2019 at 1:25 am · Reply

Hello!
On the end of the course, you've said that when we make forward declaration of a constant global variable in order to use it in another file, it should not be initialized. Shouldn't we initialize all constant variables in the moment of definition? What should I use if I want a constant global variable with external linkage and I want it to be initialized during definition?

Thank you!

Alex
August 27, 2019 at 1:27 pm · Reply

Forward declarations just tell the compiler that an object (or function) exists (and what type it is). They don't define actual objects, and so they can't have initializers.

The actual defined object (that the forward declaration is referencing) should be initialized.

SM.Haider
August 12, 2019 at 12:32 pm · Reply

Wait, what's the scope of this article?

Gabe
July 26, 2019 at 1:44 pm · Reply

"Identifiers with external linkage will generally cause a duplicate definition linker error if the definitions are compiled into more than one .cpp file (due to violating the one-definition rule)."

Would you mind explaining this sentence? What does "if the definitions are compiled into more than one .cpp file"?

nascardriver
July 27, 2019 at 11:40 pm · Reply

gabe.hpp

```
// `gabe` has external linkage.
int gabe{};
```

gabe.cpp

```
#include "gabe.hpp"
```

main.cpp

```
#include "gabe.hpp"

int main(void)
{

  return 0;
}
```

`gabe` is now defined in "gabe.cpp" and "main.cpp" (Because `#include` simply copies a file's content). Variables with external linkage can only be defined once, so you're getting and error.

murat yilmaz
August 12, 2019 at 1:40 pm · Reply

How to prevent this from happening?

nascardriver
August 12, 2019 at 11:34 pm · Reply

gabe.hpp

```
// Declare as external in the header.
extern int gabe;
```

gabe.cpp
[/CODE]
#include "gabe.hpp"

// Define in a source file
int gabe{};
[/CODE]

Samira Ferdi

June 30, 2019 at 7:17 pm · Reply

Hi, Alex and Nascardriver!

What is normal function? I confused the term 'normal'. So, are there functions those are not normal?

Alex
July 1, 2019 at 12:20 pm · Reply

Normal functions are the kind of functions we've presented so far. There are other kinds of functions (member functions, inline functions, etc...) that you'll learn about later.

Samira Ferdi
July 1, 2019 at 4:13 pm · Reply

Thank you, Alex!

What is the scope, duration, and linkage of namespace?

« Older Comments 1 2