7.11 — Scope, duration, and linkage summary

learncpp.com/cpp-tutorial/scope-duration-and-linkage-summary/

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 (local) scope** can only be accessed from the point of declaration until the end of the block in which they are declared (including nested blocks). This includes:
    - Local variables
    - Function parameters
    - Program-defined type definitions (such as enums and classes) declared inside a block
- Variables and functions with **global scope** can be accessed from the point of declaration until the end of the file. This includes:
    - Global variables
    - Functions
    - Program-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 a declaration of that same identifier in a different scope refers to the same entity (object, function, reference, etc…) or not.

Local variables have no linkage. Each declaration of an identifier with no linkage refers to a unique object or function.

- An identifier with **no linkage** means another declaration of the same identifier refers to a unique entity. Entities whose identifiers have no linkage include:
  - Local variables
  - Program-defined type identifiers (such as enums and classes) declared inside a block
  - Namespaces
- An identifier with **internal linkage** means a declaration of the same identifier within the same translation unit refers to the same object or function. Entities whose identifiers have internal linkage include:
  - Static global variables (initialized or uninitialized)
  - Static functions
  - Const global variables
  - Functions declared inside an unnamed namespace
  - Program-defined type definitions (such as enums and classes) declared inside an unnamed namespace
- An identifier with **external linkage** means a declaration of the same identifier within the entire program refers to the same object or function. Entities whose identifiers have external linkage include:
  - Functions
  - Non-const global variables (initialized or uninitialized)
  - Extern const global variables
  - Inline const global variables

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 local 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;	Global	Static	External	Initialized or uninitialized
Internal non-constant global variable	static int g_x;	Global	Static	Internal	Initialized or uninitialized
Internal constant global variable	constexpr int g_x { 1 };	Global	Static	Internal	Must be initialized
External constant global variable	extern const int g_x { 1 };	Global	Static	External	Must be initialized
Inline constant global variable (C++17)	inline constexpr int g_x { 1 };	Global	Static	External	Must be initialized

Forward declaration summary

You can use a forward declaration to access a function or variable in another file. The scope of the declared variable is as per usual (global scope for globals, block scope for locals).

Type	Example	Notes
Function forward declaration	void foo(int x);	Prototype only, no function body
Non-constant variable forward declaration	extern int g_x;	Must be uninitialized
Const variable forward declaration	extern const int g_x;	Must be uninitialized
Constexpr 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	
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.