

CSE 2421

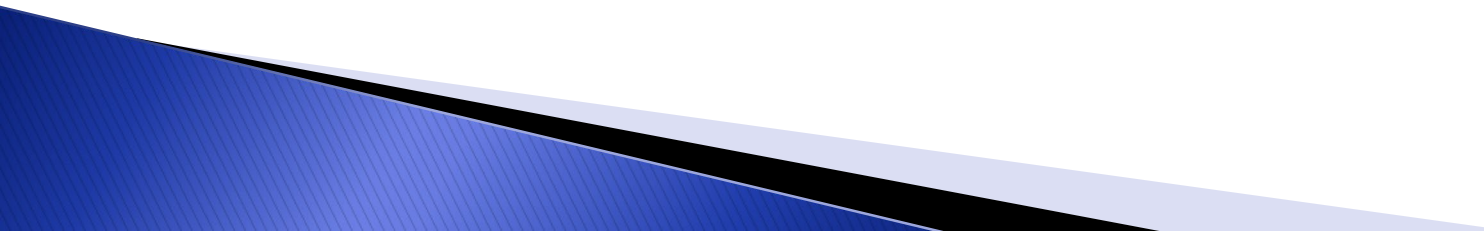
## C Storage Class, Scope, and Linkage

Recommended Reading: *C Primer Plus*, Chapter 12, sections on those 3 topics

# Overview of concepts and terms

- ▶ **Storage class: 4 types (only 3 really)**
  - *static*
  - *automatic (or auto)*
  - *register*
  - *external* (not really storage class, but linkage)
- ▶ **Storage Scope: 4 types (we will only cover 3)**
  - *block*
  - *file*
  - *prototype*
  - *function* (very limited use (only labels for goto); we will not cover this)
- ▶ **Storage Linkage: 3 types**
  - *internal*
  - *external*
  - *none* (no linkage)
- ▶ **CAUTION: Learn the terms!** “Global” and “Local” are not correct terms for storage class, scope, or linkage in C! [You will lose points if you use these on an exam.]

# Overloading the word static

- ▶ The keyword static has different meanings depending on context.
  - ▶ In this slide deck we have more than one such meaning
  - ▶ We also talk about “statically allocated arrays” sometimes shortened to “static arrays” which is a completely different use of the word
  - ▶ You need to keep all of those meanings straight
- 

# Terms/Concepts - introduction

1. **Storage class**: determines (a) **the type of storage** and **the lifetime** of the storage associated with a variable, and (b) the variable's **initialization**.
2. **Scope**: determines the region of a program in which a variable can be accessed **directly** (i.e., **by name**, and *not through a pointer*).
3. **Linkage**: for a given name (identifier), determines whether *the same name in a different scope* refers to *the same variable or function*, or *some different variable or function* [e.g. You have someone in 3 of your classes named Noah. Is it the same person or 3 different people all named Noah?]

# Storage Class

- Refers to the type of memory in which the variable's value is stored, which in turn defines different characteristics for the variable
  - static (keyword) – The heap
  - auto (keyword) – The runtime stack
  - register (keyword) - Hardware registers
- Determines when the variable is created and destroyed and how long it will retain its value. Note this information on the following slides.
- Determines whether the variable will be initialized by the compiler (the loader actually) or whether it must be done explicitly by the program code
- **The *default storage class*** for a variable depends on ***where it is declared*** (see following slides for different possible places in a C program)

# Storage Classes - Static

- Declared **outside any blocks**
  - Storage class is **static** (stored on **the heap**)
  - No way to specify any other storage class for these variables
  - These variables are created before the program begins to run (or dynamically at runtime – more later) and exist throughout its entire execution.
  - These variables are **initialized** by default (by the loader) **to a zero value** appropriate to their declared type
  - They **retain last value they were assigned until the program completes** (i.e., the values are retained from the completion of one function to the beginning of the next function)
  - Can be declared above all functions in the program, but can also be declared between two functions (this affects **scope**, though; see below)

```
#include <stdio.h>
{
    int main() {
        .....
    }
```



# Storage Classes - Automatic

- Declared **within a block**
  - Storage class default is **automatic** (stored on **the runtime stack**)
  - Created just before execution of any code in the block in which they are declared
  - “Discarded” (technically, we should say “inaccessible”) just as execution leaves that block
  - These variables **must be explicitly initialized by your code**, or they will contain **garbage** (unknown/unpredictable values)
  - New copies created each time the block is executed, so **values are not retained** in between each of multiple executions of the code in the block (for example, between function calls).
  - **Parameters** passed to functions have **automatic** storage class also (to support recursion), and there is no way to change this (see below)



```
#include <stdio.h>
int main() {
    .....
}
```

# Storage Classes – static

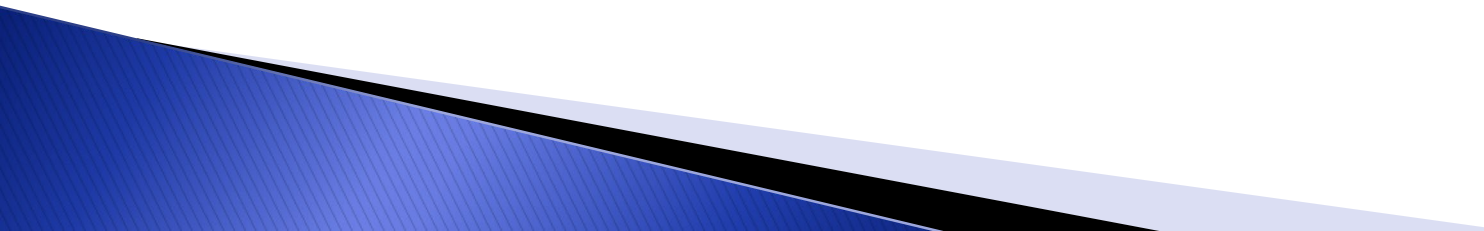
- Variables declared within a block but with the keyword ***static*** change storage class from automatic (the default) to static
- Static storage class exists for the entire duration of the program, rather than just the duration of the block in which the variable is declared
- NOTE: the changing of the storage class of a variable does not change its scope (see below); it is still accessible by name only from within the block
- NOTE: parameters to a function cannot be declared static, because arguments are always passed on the stack to support recursion. Thus, **the keyword *static* cannot be used with parameters.**



# Storage Classes – static (cont)

- When used in declarations of variables that appear outside of blocks, or in function definitions (i.e., the function is designated as static)
  - The keyword static changes **the linkage** from external to internal
  - The storage class and scope are not affected.
  - Thus, this is really **a *distinct* use of the keyword *static*** in C, because it relates to ***linkage***, and not to storage class (although historically, it was referred to as storage class)!
  - Accessible only from within the source file in which they were declared
  - This is used to limit the accessibility of “global” variables (variables declared outside any block) in library functions (so that, if you happen to use a global variable in your program with the same name as a global in the library function, you will not get unexpected results).

# Storage classes - register

- Can be used on automatic variables to indicate that they should be stored in the machine's hardware registers rather than in memory.
  - WHY? To be accessed faster
  - NOTE – the compiler can ignore this if necessary; i.e., if too many variables designated as register (first come, first served) → rest are automatic
  - Typically declare heavily used variables as register variables (i.e., loop indices)
  - Created and destroyed at the same time as automatic variables
- 

# Storage Class Summary

- Automatic
  - Default for variables declared inside blocks
  - Discarded on exit from block
  - Can have auto specifier – but it is redundant (it's the default)
  - Can have register specifier
    - Stored in fast registers of the machine *if possible* instead of RAM
- Static
  - Default for variables declared outside of any block
    - Declared outside function blocks
  - Can also be declared within a function with keyword static
    - Initialized at load time and retains its value between calls to function
    - Initial value must be a constant known at compile time

# Scope

- ▶ The area in the program in which an identifier may be used to access a variable directly (i.e., not through a pointer)
- ▶ For example, the scope of the variables declared inside a block in a function is limited to that block in that function
- ▶ This means:
  - No other function may access these variables by their names because the names cannot validly refer to these variables outside of their scope
  - It is legal to declare **different variables** with the same names so long as they are not declared in the same scope.
- ▶ Scope types:
  - block
  - file
  - prototype
  - function (only relevant to labels with goto – we will not cover this)
- ▶ Where a variable (identifier) is declared determines its scope

# Block scope

- A block is a list of statements enclosed in braces
- Any identifiers declared at the beginning of the block (where they must be in ANSI C89/C90) are accessible to all statements in the block
- The formal parameters of a function definition also have block scope in the function's body
  - Local variables declared in the outermost block cannot have the same name as any of the parameters because they are all declared in the same scope
- Nested blocks having declarations of variables with the same name
  - The outer block variable cannot be referenced by name from within the inner block, because they are hidden by the variable declared in the inner block (DO NOT DO THIS, because it decreases program readability, and is error-prone)

# Filescope

- Any identifier declared outside of all blocks
- Means that the identifier may be accessed anywhere from the end of its declaration to the end of the source file in which it was declared



# Prototype Scope

- Applies to variable (parameter) names given in a function prototype (declaration)
- Remember it's not required to supply names (only types are required)
- Prototype scope extends uniquely to each prototype (the prototype scope of one prototype is different from that of every other prototype)

*/\* Valid prototypes \*/*

```
void function1(int var1, int var2);
```

```
void function2(int var1, int var2);
```



# Linkage Types

- Linkage determines how multiple occurrences of an identifier in different scopes are treated
- The scope of a variable named by an identifier is related to its linkage, but the two properties are not the same: **scope** is the part of a program in a given source file where a variable can be accessed by its name, but linkage relates to whether occurrences of the same identifier in two different source files refer to the same variable, or different variables. For that reason, *linkage only relates to **file scope** variables and functions.*

# Linkage Types

- 3 types of linkage
  - None (no linkage)
    - Identifiers that have no linkage (variables without file scope) are always individuals; i.e. multiple declarations using the same identifier are always treated as separate and distinct entities. Block scope variables and parameters in function prototypes have no linkage.
  - Internal linkage
    - All references to the identifier within one source file refer to a single variable (the one declared in that source file), but declarations of, and references to, the same identifier in other source files refer to different variables
  - External linkage
    - All references to an identifier in any source file refer to the same entity
    - Variable known to all functions in the program; declared outside any function.

# Linkage

- After the individual source files comprising a program are compiled, the object files are linked together with functions from one or more libraries to form the executable program

# extern

- ▶ We use the extern keyword to *declare* a file scope variable when we don't want to *define* it
- ▶ Can help assure that there is a single definition for a variable

```
int key=0; /* this defines (and thus declares) key */
```

```
extern int key; /* this declares key - has to be defined elsewhere */
```



# Example: Block scope, static storage, no linkage variable

```
void print_count()  
{  
    /* the init happens once, at load time */  
    static int private_count = 0;  
  
    private_count++;  
    printf("This function has been called %d  
times\n", private_count);  
}
```