CC-112 Programming Fundamentals

Storage Classes and Scope Rules

Nazar Khan
Department of Computer Science
University of the Punjab

Storage Classes

- Each identifier in a program has the attributes
  1. storage class
  2. storage duration – <u>when</u> does an identifier exists in memory?
  3. scope – <u>where</u> can an identifier be referenced in a program?
  4. linkage – <u>which</u> other C files can reference an identifier?
- C provides four *storage classes* indicated by the storage class specifiers:
  1. `auto`,
  2. `register`,
  3. `extern`, and
  4. `static`.
- An identifier's *storage duration* is when that identifier exists in memory.
- An identifier's *linkage* determines for a multiple-source-file program whether an identifier is known only in the current source file or in any source file with proper declarations.

Storage Classes

- Variables with automatic storage duration are created when the block in which they're defined is entered, exist while the block is active and are destroyed when the block is exited. A function's local variables normally have automatic storage duration.
- Keywords `extern` and `static` are used to declare identifiers for variables and functions of static storage duration.
- Static storage duration variables are allocated and initialized once, before the program begins execution.
- There are two types of identifiers with static storage duration:
  1. external identifiers (such as global variables and function names), and
  2. local variables declared with the storage-class specifier static.

Global vs Local Static

- Global variables are created by placing variable definitions outside any function definition.
- Global variables retain their values throughout the execution of the program.
- Local static variables retain their value between calls to the function in which they're defined.
- All numeric variables of static storage duration are initialized to zero if you do not explicitly initialize them.

Scope Rules

- An identifier's scope is where the identifier can be referenced in a program.
- An identifier can have
  1. function scope,
  2. file scope,
  3. block scope, or
  4. function-prototype scope.

Function scope

- Labels are the only identifiers with function scope.
- Labels can be used anywhere in the function in which they appear but cannot be referenced outside the function body.

File scope

- An identifier declared outside any function has file scope.
- Such an identifier is “known” in all functions from the point at which it’s declared until the end of the file.

Block scope

- Identifiers defined inside a block have block scope.
- Block scope ends at the terminating right brace (}) of the block.
- Local variables defined at the beginning of a function have block scope.
- Function parameters are considered local variables by the function and also have block scope.
- Any block may contain variable definitions. When blocks are nested, and an identifier in an outer block has the same name as an identifier in an inner block, the identifier in the outer block is "hidden" until the inner block terminates.

Function-prototype scope

- The only identifiers with function-prototype scope are those used in the parameter list of a function prototype.
- Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity.

Example of scoping rules

```c
// Scoping.
#include <stdio.h>

void useLocal(void); // function prototype
void useStaticLocal(void); // function prototype
void useGlobal(void); // function prototype
int x = 1; // global variable

int main(void)
{
    int x = 5; // local variable to main
    printf("local x in outer scope of main is %d\n", x);
    { // start new scope
        int x = 7; // local variable to new scope
        printf("local x in inner scope of main is %d\n", x);
    } // end new scope
    printf("local x in outer scope of main is %d\n", x);
    useLocal(); // useLocal has automatic local x
    useStaticLocal(); // useStaticLocal has static local x
    useGlobal(); // useGlobal uses global x
    useLocal(); // useLocal reinitializes automatic local x
    useStaticLocal(); // static local x retains its prior value
    useGlobal(); // global x also retains its value
    printf("\nlocal x in main is %d\n", x);
}

// useLocal reinitializes local variable x during each call
void useLocal(void)
```

Example of scoping rules

```
{
    int x = 25; // initialized each time useLocal is called
    printf("\nlocal x in useLocal is %d after entering useLocal\n", x);
    ++x;
    printf("local x in useLocal is %d before exiting useLocal\n", x);
}

// useStaticLocal initializes static local variable x only the first time
// the function is called; value of x is saved between calls to this
// function
void useStaticLocal(void)
{
    // initialized once
    static int x = 50;
    printf("\nlocal static x is %d on entering useStaticLocal\n", x);
    ++x;
    printf("local static x is %d on exiting useStaticLocal\n", x);
}
// function useGlobal modifies global variable x during each call
void useGlobal(void)
{
    printf("\nglobal x is %d on entering useGlobal\n", x);
    x *= 10;
    printf("global x is %d on exiting useGlobal\n", x);
}
```

Example of scoping rules

Produces the following output

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```