Storage Classes

ESC108A Elements of Computer Science and Engineering

B. Tech. 2017

Course Leaders:

Roopa G.

Ami Rai E.

Chaitra S.

Objectives

- At the end of this lecture, student will be able to
  - identify the scopes of variables declared in C programming language

Contents

- Local and global variables
- Automatic variables
- External variables
- Static variables
- Register variables

Scope of Variables

- The scope of a variable determines over what part(s) of the program a variable is actually available for use(active)
  1. Local(internal) variables
     - Declared within a particular function
     - All variables declared inside functions are local to that function
  2. Global(external) variables
     - Declared outside any function

- In case, local variable and global variable have the same name, the local variable will have precedence over the global one

Local and Global Variables - Example

```
void test();
int x=66;  //x is a global variable
int main(int argc, char** argv) {
    int m=22,n=44; // m, n are local variables of main function
    printf("\nvalues : m=%d,n=%d,x=%d",m,n,x);
    test();
    return (EXIT_SUCCESS);
}
void test(){
    int a=50,b=80; // a, b are local variables of test function
    printf("\nvalues : a=%d,b=%d,x=%d",a,b,x);
}
```

Global Variable Example

```
int x;
int main() {
    x=10;
    printf("x=%d\n",x);
    printf("x=%d\n",fun1());
    printf("x=%d\n",fun2());
    printf("x=%d\n",fun3());
 }
int fun1() {
     x=x+10;
    return(x);
 }
```

```
int fun2(){
    int x;
    x=1;
    return(x);
 }
int fun3(){
    x=x+10;
    return(x);
 }
```

Output

x=10

x=20

x=1

x=30

Storage Classes

- An identifier's storage class
  - determines its storage duration, scope and linkage

- An identifier's storage duration
  - period during which the identifier exists in memory
- An identifier's scope
  - where the identifier can be referenced in a program
- An identifier's linkage
  - determines whether the identifier is known only in the current source file or in any source file with proper declarations

Storage Specifiers

- Only one storage specifier can be applied to one identifier
- Storage specifiers
  1. auto
  2. extern
  3. static
  4. Register

Storage Durations

The four storage-class specifiers can be split into two storage durations

1. Automatic storage duration
   - created when the block in which they're defined is entered; they exist while the block is active, and they're destroyed when the block is exited
   - auto and register
   - Only variables
2. Static storage duration
   - Exist from the point at which the program begins execution
   - extern and static

Automatic Variables

- Declared inside a function in which they are to be utilized
  - Created when a function is called and destroyed automatically when the function is exited
- Keyword: auto
- Example

```
auto int number;
```

- Variables declared inside a function without storage class specification is, by default, an automatic variable
- If automatic variables are not initialized they will contain garbage

Automatic Variables - Example

```
int main(){
     int m=1000;
    function2();
    printf("%d\n",m);
}
function1(){
      int m = 10;
      printf("%d\n",m);
}
function2(){
    int m = 100;
    function1();
    printf("%d\n",m);
}
```

Output
10
100
1000

Register Variables

- Normally variables are stored in the main memory
- But some programs may need variables whose values are stored in the CPU registers
- Declared using register keyword
- Example

  register int count;

- Faster execution of program
  - Register access are much faster than a memory access
  - keep frequently accessed variables in the register

Register Variables contd.

- It is important to carefully select the variables for this purpose
  - Only few variable can be placed in the register
- C will automatically convert register variables into nonregister variables once the limit is reached
- Don't try to declare a global variable as register
  - register will be occupied during the lifetime of the program

External Variables

- Variables that are both alive and active throughout the entire program
- Global variables and function names are of storage class extern by default

- Default value is zero

- Keyword: **extern**
- Example : extern int y;

External Declaration

```
int main(){
    y=5;
    . . .
}
int y;

func1(){
    y=y+1;
}
```

- As far as main is concerned, y is not defined. So compiler will issue an error message.
- There are two way out at this point
  1. Define y before main.
  2. Declare y with the storage class ***extern*** in main before using it.

External Declaration - Example

```
extern int var;
int main(void){
    var = 10;
    return 0;
}
```

- This program throws error in compilation
- *var* is declared but not defined anywhere
- the var isn't allocated any memory

```
#include "somefile.h"
extern int var;
int main(void){
    var = 10;
    return 0;
}
```

- Supposing that somefile.h has the definition of var
- This program will be compiled successfully

```
extern int var = 0;
int main(void){
    var = 10;
    return 0;
}
```

- if a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined
- So it will work

Static Variables

- The value of static variables persists until the end of the program
- Static variables are initialized only once, when the program is compiled

- if nothing is initialised, by default, it is assigned zero

- It is declared using the keyword ***static***

```
static int x;
```

Static Variable - Example

- static variable can be used to count the number of calls made to function

```
int main(){
    int i;
    for(i =1; i<=3; i++)
    stat();
 }
void stat(){
    static int x=0;
     x = x+1;
     printf("x = %d\n",x);
}
```

```
Output
x=1
x=2
x=3
```

Scope Rules

- The scope of an identifier
  - the portion of the program in which the identifier can be referenced

- The four identifier scopes
  1. function scope
  2. file scope
  3. block scope
  4. function-prototype scope

Scope Rules contd.

1. Function scope
   - Labels (an identifier followed by a colon such as start:)
   - Labels can be used anywhere in the function in which they appear, but cannot be referenced outside the function body
   - Labels are implementation details that functions hide from one another
2. File scope
   - An identifier declared outside any function
   - identifier is “known” (i.e., accessible) in all functions from the point at which the identifier is declared until the end of the file
   - Global variables, function definitions, and function prototypes placed outside a function

Scope Rules contd.

3. Block scope
   - Identifiers defined inside a block
   - Block scope ends at the terminating right brace (}) of the block
   - Local variables defined at the beginning of a function, function parameters which are considered local variables by the function, Local variables declared static
4. function-prototype scope
   - The identifiers those are used in the parameter list of a function prototype
   - Ignored by the compiler
   - Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity

Scope Rules - Example

```
#include <stdio.h>
int cube( int y );
int main( void ){
        int x;
        for ( x = 1; x <= 10; x++ )
            printf( "%d\n", cube( x ));
        return 0;
}
int cube( int y ){
        return y * y * y;
}
```

- Block scope
  - The variable x in main
  - The variable y in cube
- File scope
  - The function cube
  - The function main
  - The function prototype for cube
- Function prototype scope
  - The identifier y in the function prototype for cube

Summary

- The scope of a variable determines over what part(s) of the program a variable is actually available for use(active)
- An identifier's storage class determines its storage duration, scope and linkage
- Only one storage specifier can be applied to one identifier

Further Reading

Kernighan, B. W. and Richie, D. (1992) *The C Programming Language.* 2nd ed., New Delhi:PHI.