Storage classes

- . There are two different ways to characterize variables: by ***data type***, and by ***storage class***
- Data type refers to the type of information represented by a variable, e.g., integer number, floating-point number, character, etc.
- Storage class refers to the permanence of a variable, and its ***scope*** within the program,
- The storage class of a variable in C determines
  - the life time of the variable if this is 'global' or 'local'.
  - storage class also determines variable's storage location (memory or registers),
  - the scope (visibility level) of the variable,
  - and the initial value of the variable
- There are four different storage-class specifications in C: ***automatic***, ***external***, ***static*** and ***register***.
- are identified by the keywords **auto, extern, static , and register** , respectively.

These specifiers tell the compiler how to store the subsequent variable.

The general form of a variable declaration that uses a storage class is shown here:

storage_class_specifier data_type variable_name;
Example:

```
auto int a, b, c;

extern float root1, root2;

static int count = 0;

extern char star;
```

Automatic Variables

- A variable defined within a function or block with auto specifier belongs to automatic storage class.
- All variables defined within a function or block by default belong to automatic storage class
- Variables having automatic storage class are local to the block which they are defined in, and get destroyed on exit from the block.
- No default value will be assigned to Automatic variables, unless it is initialized

```c
#include <stdio.h>
int main( )
{
  auto int i = 1;
  {
    auto int i = 2;
    {
      auto int i = 3;
      printf ( "\n%d ", i);
    }
    printf ( "%d ", i);
  }
  printf( "%d\n", i);
}
```

OUTPUT

```
======
3 2 1
```

AUTOMATIC VARIABLES -in functions

- Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.
- Since the location of the variable declarations within the program determines the automatic storage class, the keyword **auto** is not required at the beginning of each variable declaration.

```
include <stdio.h>
long int factorial(int n);

main()
{
     auto int n;
     /* read in the integer quantity */
     printf("\nn = ");
     scanf("%d", &n);
     /* calculate and display the factorial */
     printf("\n%d! = %ld", factorial(n));
}

long int factorial(auto int n)          /* calcu
{
     auto int i;
     auto long int prod = 1;
     if (n > 1)
        for (i = 2; i <= n; ++i)
             prod *= i;
     return(prod);
}
```

Scope

- ***An automatic variable does not retain its value once control is transferred out of its defining function.***
- Therefore, **any** value assigned to **an** automatic variable within a function will be lost once the function is exited.

EXTERNAL (GLOBAL) VARIABLES

- ***External variables***, in contrast to automatic variables, are not confined to single functions.
- Their scope extends from **the point of definition** through the remainder of the program.
- Hence, they usually span two or more functions, and often an entire program. They are often referred to as ***global variables.***
- The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program
- The use of external variables provides a convenient mechanism for transferring information back and forth between functions.
    - transfer information into a function without using arguments.
    - transfer multiple data items out of a function

External Variable definition

- An external variable ***definition*** **is** written in the same manner **as** an ordinary variable declaration.
- It must appear outside of, and usually before, the functions that access the external variables.
- The assignment of initial values can be included within an external variable definition if desired.
- The storage-class specifier extern is not required in an external variable definition.

- External variables can be assigned initial values(CONSTANTS) **as** a part of the variable definitions.
- The external variables will then retain these initial values unless they are later altered during the execution of the program.
- Ie, Alteration in the value of an external variable within a function will be carried over into other parts of the program.
- If an initial value is not included in the definition **of** an external variable, the variable will automatically be assigned a value of zero.

```
#include <stdio.h>
    int x; // external variable definition
    int main()
{
  printf("x: %d\n", x);
}
```

Default value is 0

```
#include <stdio.h>
    int x =2; // external variable definition
     void changex();
    int main()
{
  printf("x: %d\n", x);
  x=x+1;
 printf("x: %d\n", x);
changex();
printf("x: %d\n", x);
}
changex()
{
x= 0;
printf("x: %d\n", x);
return;
}
```

Output:
x: 2
x: 3
x: 0
x: 0

External variable declaration

- A declaration declares the name and type of a variable or function.
- A definition causes storage to be allocated for the variable or the body of the function to be defined.

External variable declaration

- If a function definition ***precedes*** the external variable definition, then the function must include a ***declaration*** for that external variable .
- An external variable ***declaration*** must begin with the storage-class specifier **extern.** The name of the external variable and its data type must agree with the corresponding external variable definition that appears outside of the function.
- Storage space for external variables will ***not*** be allocated **as** a result of an external variable declaration.
- Moreover, an external variable declaration ***cannot*** include the assignment of initial values.
- These distinctions between an external variable ***definition*** and an external variable ***declaration*** are very important.

```
#include <stdio.h>
    extern int x; // external variable declaration
    int main()
{
  printf("x: %d\n", x);
}
 int x = 10;       //external variable definition
```

Case 1:
if you remove extern int x;
you will get an error *"Undeclared identifier ‘x’*
Case 2:
change the statement extern int x; to extern int x = 50;
you will get an error *"Redefinition of 'x'*

Note that extern can also be applied to a function declaration,
but doing so is redundant because all function declarations are implicitly extern

Static storage Class

- The static specifier gives the declared variable static storage class.
- Static variables can be used within function or file.
- Unlike global variables, static variables are not visible outside their function or file, but they maintain their values between calls.
- Static variables have default initial value zero and initialized only once in their lifetime.
- The static specifier has different effects upon local and global variables

- When static specifier is applied to a local variable inside a function or block, retains its value between function calls

```c
#include <stdio.h>

void staticDemo()
{
  static int i;
  {
    static int i = 1;
    printf("%d ", i);
    i++;
  }
  printf("%d\n", i);
  i++;
}

int main()
{
  staticDemo();
  staticDemo();
}
```

```
OUTPUT
======
1 0
2 1
```

- When static specifier is applied to a global variable or a function then compiler makes that variable or function known only to the file in which it is defined.

```c
/* staticdemo.c */
#include <stdio.h>
static int gInt = 1;
static void staticDemo()
{
   static int i;
   printf("%d ", i);
   i++;
   printf("%d\n", gInt);
   gInt++;
}

int main()
{
   staticDemo();
   staticDemo();
}
OUTPUT
======
0 1
1 2
```

Static Variables in a single file program (entire program is contained within a single source file)

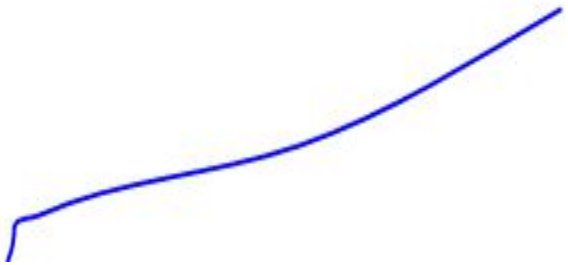
- Static variables are defined within individual functions and therefore have the same scope as automatic variables.
- i.e., they are local to the functions in which they are defined.(cannot, however, be accessed outside of their defining function.)
- Static variables retain their values throughout the life of the program.
- Variable declaration must begin with the **static** storage-class designation.

Only b and c is recogonized as external variable in main()

```
float a, b, c;

void dummy(void);

main()
{
     static float a;

     . . . . .
}

void dummy(void)
{
     static int a;
     int b;

     . . . . .
}
```

Rules associated with the assignment of Static values

1. The initial values must be expressed as constants, not expressions.
2. The initial values are assigned to their respective variables at the beginning of program execution. The variables retain these values throughout the life of the program, unless different values are assigned during the course of the computation.
3. Zeros will be assigned to all static variables whose declarations do not include explicit initial values. Hence, static variables will always have assigned values.



Static Variables in a multi file program

- Rules associated with the use of functions

1. Within a multifile program, a function definition may be either ***external*** or ***static***.
2. An external definition will be recognized throughout the entire program, whereas a static function will be recognized only within the file in which it is defined.
3. In each case, the storage class is established by placing the appropriate storage-class designation (i.e., either **extern** or **static)** at the beginning of the function definition.
4. The function is assumed to be ***external*** if a storage class designation does not appear.

Function definition syntax

```
storage-class  data-type  name(type 1  arg 1,  type 2  arg 2, . . .,
                                                        type n  arg n)
```

Function declaration

- When **a** function is defined in one file and accessed in another, the latter file must include a function ***declaration***. This declaration identifies the function **as** an external function whose definition appears elsewhere.

In general terms, a function *declaration* can be written as

```
storage-class  data-type  name(argument type 1,  argument type 2, . . .,
                                                        argument type n);
```

A function declaration can also be written using full function prototyping (see Sec. 7.4) as

```
storage-class  data-type  name(type 1  arg 1,  type 2  arg 2, . . .,
                                                    type n  arg n);
```

To execute a multifile program, each individual file must be compiled and the resulting object files linked together.

First file:

```
/* simple, multifile program to write "Hello, there!" */

#include <stdio.h>

void output(void);      /* function prototype */

main()
{
    output();
}
```

Second file:

```
void output(void)      /* external function definition */
{
    printf("Hello, there!");
    return;
}
```

Rules for variables (Definition)

- Within a multifile program, external (global) variables can be defined in one file and accessed in another.
- An external variable ***definition*** can appear in only one file. Its location within the file must be external to any function definition.
- Usually, it will appear at the beginning of the file, ahead of the first function definition.
- The storage-class specifier **extern** is not required within the variable definition

Rules for variables (Declaration)

- In order to access an external variable in another file, the variable must first be ***declared*** within that file.
- This declaration may appear anywhere within the file. Usually, however, it will be placed at the beginning **of** the file, ahead of the first function definition.
- The declaration ***must*** begin with the storage-class specifier **extern.**
- Initial values ***cannot*** be included in external variable declarations.

First file:

```
int a = 1, b = 2, c = 3;      /* external variable DEFINITION */

extern void funct1(void);     /* external function DECLARATION */

main()                        /* function DEFINITION */
{
    . . . . .
}
```

Second file:

```
extern int a, b, c            /* external variable DECLARATION */

extern void funct1(void)      /* external function DEFINITION */
{
    . . . . .
}
```

REGISTER VARIABLES

- Registers are special storage areas within the computer's central processing unit.
- The actual arithmetic and logical operations that comprise a program are carried out within these registers.
- Normally, these operations are carried out by transferring information from the computer's memory to these registers, carrying out the indicated operations, and then transferring the results back to the computer's memory.
- This general procedure is repeated many times during the course of a program's execution.

- For some programs, the execution time can be reduced considerably if certain values can be stored within these registers rather than in the computer's memory.
- Such programs may also be somewhat smaller in size (i.e., they may require fewer instructions), since fewer data transfers will be required

- In C, the values of ***register variables*** are stored within the registers of the central processing unit.
- A variable can be assigned this storage class simply by preceding the type declaration with the keyword register .
- There can, however, be only a few register variables (typically, two or three) within any one function.
- The exact number depends upon the particular computer, and the specific C compiler. Usually, only integer variables are assigned the register storage class.

- The register and automatic storage classes are closely related. In particular, their visibility (i.e., their scope) is the same.
- Thus, register variables, like automatic variables, are local to the function in which they are declared. Furthermore, the rules governing the use of register variables are the same as those for automatic variables , except that the address operator *(&)* cannot be applied to register variables.

- Moreover, declaring certain variables to be register variables does not guarantee that they will be actually be treated as register variables.
- The declaration will be valid only if the requested register space is available.
- If a register declaration is not honored, the variables will be treated as having the storage class automatic.

```
#include <stdio.h>

int main()
{
  register int i = 10;
  int *p = &i; //error: address of register variable requested

  printf("Value of i: %d", *p);
  printf("Address of i: %u", p);
 }
```

Output:

Error:address of register variable requested