Passing Parameters to Functions

There are two ways of passing parameters to the functions.

1. call by value
2. call by reference

call by value

When a function is called with actual parameters, the values of actual parameters are copied into the formal parameters. If the values of the formal parameters changes in the function, the values of the actual parameters are not changed. This way of passing parameters is called call by value (pass by value).

In the below example, the values of the arguments to swap () 10 and 20 are copied into the parameters x and y. Note that the values of x and y are swapped in the function. But, the values of actual parameters remain same before swap and after swap.
Note: In call by value any changes done on the formal parameter will not affect the actual parameters.

```
// C program illustrates call by value
#include<stdio.h>
void swap (int , int );
void main ()
{
  int a, b;
  clrscr ();
  printf (“\n Enter the values of a and b:  “);
  scanf (“%d%d”, &a, &b);
  swap (a, b);    /*calling function */
  printf (“\nFrom main The Values of a and b a=%d, b=%d “, a, b);
  getch ();
}
void swap (int x, int y)
{
  int temp;
  temp=x;
  x=y;
  y=temp;
  printf (“\n The Values of a and b after swapping a=%d, b =%d”, x, y);
}
```

OUTPUT

```
Enter the values of a and b:  10 20
The Values of a and b after swapping a=20, b=10
from main The values of a and b a=10, b=20
```

Call by Reference

Instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference", since we are referencing the variables.

Here the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Here The formal parameters should be declared as pointer variables to store the address.

The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now swapped when the control is returned to main function.

```
#include <stdio.h>
void swap ( int *pa, int *pb ) ;
int main ( )
{
  int a = 5, b = 6;
  printf("a=%d b=%d\n",a,b) ;
  swap (&a, &b) ;
  printf("a=%d b=%d\n",a,b) ;
  return 0 ;
}
```

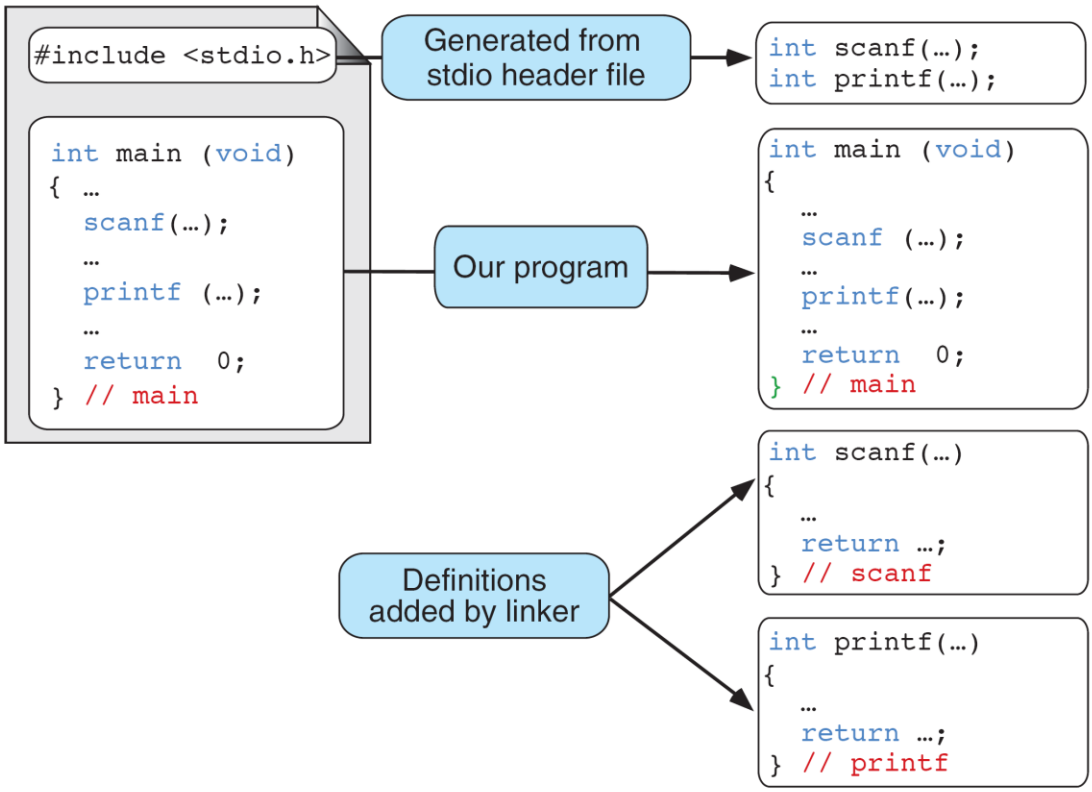
```
void swap( int *pa, int *pb )
{
  int temp;
  temp= *pa; *pa= *pb;  *pb = temp ;
  printf ("a=%d  b=%d\n", *pa, *pb);
}
```

Results:

```
a=5  b=6
a=6  b=5
a=6  b=5
```

STANDARD LIBRARY FUNCTIONS

C has a large set of built-in functions that perform various tasks. In order to use these functions their prototype declarations must be included in our program.



The above figure shows how two of the C standard functions that we have used several times are brought into out program. The include statement causes the library header file for standard input and output(stdio.h) to be copied in to our program It contains the declaration for printf and scanf.Then,when the program is linked, the object code for these functions is combined with our code to build the complete program.
Some of the header files includes these functions are

<stdio.h> Standard I/O functions
<stdlib.h> Utility functions such as string coversion routines, memory allocation routines, etc..,
<string.h> String manipulation functions
<math.h> Mathematical functions
<ctype.h> Character testing and conversion functions

NESTING OF FUNCTIONS

C permits nesting of functions, main can call function1, which calls function2, which calls function3 .., and there is no limit as how deeply functions can be nested.

Consider the following example:
#include<stdio.h>
int read ();
int sum (int, in t);
void main ()

```
{
  int  a, b;
 printf (“%d”, sum ());
  }
int sum (int x, int y)
{
   x=read ();
   y=read ();
   return x+y;
}
int read ()
{
  int p;
  printf (“\n Enter any value: “);
  scanf (“%d”, &p);
   return p;
}
```

In the above example ,when the main() function executes it finds the function call sum(), then the control is transferred from main() to the function sum(),here we are calling the function read(), then the control transferred to read() function, then the body of the function read() executes, the control transferred from read to sum() and once again the same is done for reading some other value. Then the addition is performed this value is carried from sum() to main().Observe the chain of control transfers between the nested functions.

RECURSION IN C

In C, functions can call themselves .A function is recursive if a statement of the function calls the function that contains it. This process is sometimes called circular definition.
Recursion is a repetitive process, where the function calls itself.

Concept of recursive function:

- A recursive function is called to solve a problem
- The function only knows how to solve the simplest case of the problem. When the simplest case is given as an input, the function will immediately return with an answer.
- However, if a more complex input is given, a recursive function will divide the problem into 2 pieces: a part that it knows how to solve and another part that it does not know how to solve. The part that it does not know how to solve resembles the original problem, but of a slightly simpler version.
- Therefore, the function calls itself to solve this simpler piece of problem that it does now know how to solve. This is what called the recursion step.
- The statement that solves problem is known as the base case. Every recursive function must have a base case. The rest of the function is known as the general case.
- The recursion step is done until the problem converges to become the simplest case.

- This simplest case will be solved by the function which will then return the answer to the previous copy of the function.
- The sequence of returns will then go all the way up until the original call of the function finally return the result.

Example: Recursive factorial function

Iteration definition

fact (n) =1 if n=0
=n*(n-1)*(n-2)…….3*2*1 if n>0

Recursion definition

fact (n) =1 if n=0 **BASE CASE**

=n*fact (n-1) if n>0 **GENERAL CASE**

//C program to find the factorial using recursion

```
#include <stdio.h>
long factorial (long);
void main (void)
{
    int i;
    i=4;
    printf ("%2d! = %1d\n", i, factorial (i));
}
long factorial (long number)
{
    if (number ==0)
        return 1;
    else
        return (number * factorial (number-1));
```

Output:

4! = 24

designing a recursive function:

In the above program, once the base condition has reached, the solution begins. The program has found one part of the answer and can return that part to the next more general statement. Thus, after calculating factorial (0) is 1, and then it returns 1.That leads to solve the next general case,

factorial (1) → 1*factorial (0) → 1*1 → 1

The program now returns the value of factorial (1) to next general case, factorial (2),

factorial (2) → 2*factorial (1) → 2*1 → 2

As the program solves each general case in turn, the program can solve the next higher general case, until it finally solves the most general case, the original problem.

The following are the rules for designing a recursive function:
1. First, determine the base case.
2. Then, determine the general case.
3. Finally, combine the base case and general case in to a function.

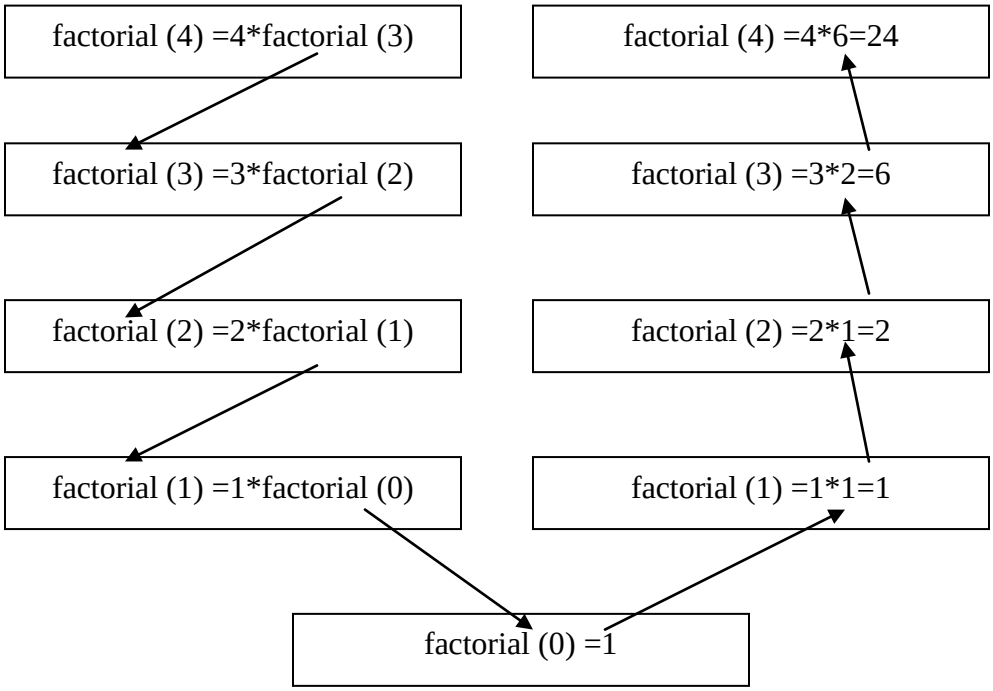


Figure: factorial (4) recursively

Difference between Iteration and Recursion

ITERATION	RECURSION
Iteration explicitly uses repetition structure.	Recursion achieves repetition by calling the same function repeatedly.
Iteration is terminated when the loop condition fails	Recursion is terminated when base case is satisfied.
May have infinite loop if the loop condition never fails	Recursion is infinite if there is no base case or if base case never reaches.
Iterative functions execute much faster and occupy less memory space.	Recursive functions are slow and takes a lot of memory space compared to iterative functions

Table: Difference between Iteration and Recursion

PREPROCESSOR COMMANDS

The C compiler is made of two functional parts: a preprocessor and a translator. The preprocessor is a program which processes the source code before it passes through the compiler. The translator converts the C statements into machine code that in places in an object module.

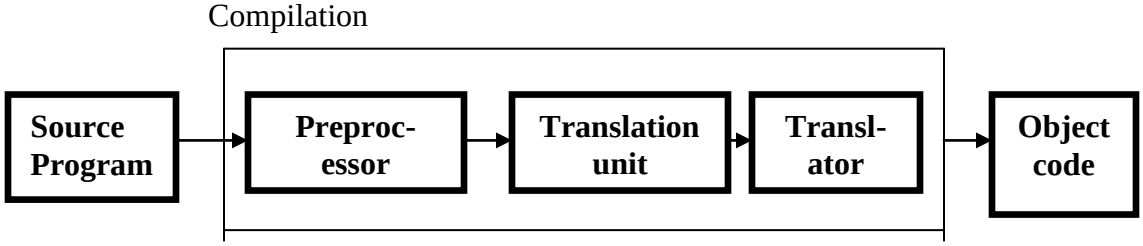


Figure 3.11 Compilation Components

The preprocessor is a collection of special statements called commands or preprocessor directives. Each directive is examined by the preprocessor before the source program passes through the compiler.

- Statements beginning with # are considered preprocessor directives.
- Preprocessor directives indicate certain things to be done to the program code prior to compilation.
- It includes certain other files when compiling, replace some code by other code.
- Only white space characters before directives on a line.

- Preprocessor commands can be placed anywhere in the program.

There are three major tasks of a preprocessor directive

1. Inclusion of other files (file inclusion)
2. Definition of symbolic constants and macros(macro definition)
3. Conditional compilation of program code/Conditional execution of preprocessor directives

FILE INCLUSION

The first job of a preprocessor is file inclusion, the copying of one or more files into programs. The files are usually header files and external files containing functions and data declarations.

General form is,

#include filename

it has two different forms

1. #include <filename>

- it is used to direct the preprocessor to include header files from the system library.
- in this format , the name of the header file is enclosed in **pointed brackets**.
- Searches standard library only for inclusion of a file.

Example:

#include<stdio.h>

In the above example ,the preprocessor directive statements searches for content of entire code of the header file stdio.h in the standard library ,if it finds there includes the contents of the entire code at the place where the statement is in the source program before the source program passes through the compiler.

Note : here we can't include user files why because it searches for the files in the standard library only.

2. #include "filename"

- it is used to direct the preprocessor look for the files in the current working directory, standard library
- Use for inclusion of user-defined files, also includes standard library files.
- Searches current directory, then standard library for the inclusion of a file.
- in this format , the name of the file is enclosed in **double quotes.**

- we can also include macros. The included file or the program either may contain main() function but not the both.

Example:

#include "header.h"

In the above example, the preprocessor directive statement directs the preprocessor to include content of entire code of the file header.h which is in the current directory at the place where the statement is in the source program before the source program passes through the compiler.

note : We can also include file that is not present in the current directory by specifying the complete path of the file like

#include "e:\muc.h"

MACRO DEFINITION

A macro definition command associates a name with a sequence of tokens. The name is called the macro name and the tokens are referred to as the macro body.

The following syntax is used to define a macro:

#define macro_name(<arg_list>) macro_body

#define is a define directive, macro_name is a valid C identifier.macro_body is used to specify how the name is replaced in the program before it is compiled.

- Example

#define CIRCLE_AREA(x) (PI * (x) * (x))

would cause

area = CIRCLE_AREA(4);

to become

area = (3.14159 * (4) * (4));

- Use parenthesis for coding macro_body,Without them the macro

#define CIRCLE_AREA(x) PI * (x) * (x)

would cause

area = CIRCLE_AREA(c + 2);

to become

area = 3.14159 * c + 2 * c + 2;

- We can also supply Multiple arguments like

#define RECTANGLE_AREA(x, y) ((x) * (y))

would cause

rectArea = RECTANGLE_AREA(a + 4, b + 7);

to become

rectArea = ((a + 4) * (b + 7));

//C program illustrating usage of Macro

```
#include<stdio.h>
#define square(x) (x*x)
int main()
{
 int a=10;

 printf("\nThe square of %d=%d",a,square(a));
  return 0 ;
}
```

OUTPUT
The square of 10=100

Symbolic Constants

Macro definition without arguments is referred as a constant. The body of the macro definition can be any constant value including integer, float, double, character or string. However, character constants must be enclosed in single quotes and string constants in double quotes.

Example

#define PI 3.14159

Here “PI” replaces with "3.14159"

Nested Macros

It is possible to nest macros. C handles nested macros by simply rescanning a line after macro expansion. Therefore, if an expansion results in a new statement with a macro, the second macro will be properly expanded.

For example,

```
#define sqre(a)  (a*a)
#define cube(a)  (sqre(a)*a)
```

The expansion of

```
x=cube(4);
```

results in the following expansion:

```
x=(sqre(4)*4);
```

this after rescanning becomes

```
x=((4*4)*4);
```

Undefining Macros

Once defined, a macro command cannot be redefined. Any attempt to redefine it will result in a compilation error. However, it is possible to redefine a macro by first undefining it, using the #undef command and the defining it again.

Example

```
#define Val 30
…
#undef Val

#define val 50
```

SCOPE

Scope determines the region of the program in which a defined object is visible. That is, the part of the program in which we can use the object's name. Scope pertains to any object that can be declared, such as a variable or a function declaration.

In C, an object can have three levels of scope

1. Block scope
2. File scope
3. Function scope

Scope Rules for Blocks

- Block is zero or more statements enclosed in a set of braces.
- Variables are in scope from their point of declaration until the end of the block. Block scope also referred as local scope.
- Variables defined within a block have a local scope. They are visible in that block scope only. Outside the block they are not visible.
- For example, a variable declared in the formal parameter list of a function has block scope, and active only in the body only. Variable declared in the initialization section of a for loop has also block scope, but only within the for statement.

```
{

        int a = 2;                      /* outer block a */
        printf ("%d\n", a);             /* 2 is printed */
        {
                int a=5;                /* inner block a */
                printf ("%d\n", a);     /* 5 is printed */
        }
        printf ("%d\n", a);             /* 2 is printed */
}

/* a no longer defined */
                                        Outer" a "Masked
```

- In the above example the variable a is created with value 2 and its scope limit to the block and it is local to that block. Here we are creating the same identifier with another value 5 in the inner block, 5 is displayed on the screen after executing inner printf statement. The a value 2 is once again in the outer block. After the block a is not visible.
- A variable that is declared in an outer block is available in the inner block unless it is redeclared. In this case the outer block declaration is temporarily "masked".
- Avoid masking! Use different identifiers instead to keep your code debuggable!

Scope rules for functions

- Variables defined within a function (including main) are local to this function and no other function has direct access to them!
- The only way to pass variables to a function is as parameters.
- The only way to pass (a single) variable back to the calling function is via the **return** statement.
- In the function scope global variable and pointer variable are exceptions, and visible here.

Scope rules for files

- File scope includes the entire source file for a program, including any files included in it.
- An object with file scope has visibility through the whole source file in which it is declared.
- Objects within block scope are excluded from file scope unless specifically declared to have file scope; in other words, by default block scope hides objects from file scope.
- File scope generally includes all declarations outside function and all function headers.
- An object with file scope sometimes referred to as a global object.
- For Example, a variable declared in the global section can be accessed from anywhere in the source program.

Local Variables

- Variables that are declared in a block known as local variables. These variables may be any variable in a function also.
- They are known in the block where they are created. Active in the block only.
- They hold their values during the execution of the block. After completing the execution of the block they are undefined.

Global Variables

- Global variables are known throughout the entire program and may be used in any piece of code.
- Also, they will hold their values during the entire execution of the program.
- Global variables are created by declaring them outside of the function and they can be accessed by any expression.

STORAGE CLASSES

Each and every variable is storing in the computer memory. Every variable and function in C has two attributes:

type (**int,float,...**)
storage class

Storage class specifies the scope of the object. It also provides information about location and visibility of the variable.

Object Storage Attributes

The storage class specifies control three attributes of an object's storage as shown in below figure 3.13. Its, scope, extent, and linkage.

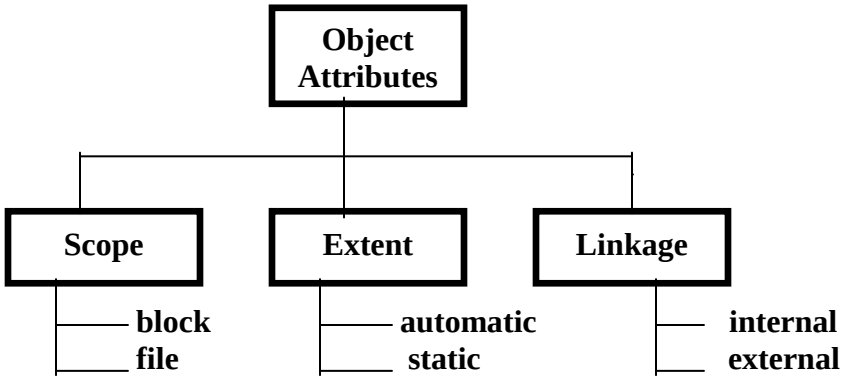


Figure: Object Storage Attributes

Scope

We have discussed this one in the previous section.

Extent

The **extent** of an object defines the duration for which the computer allocates memory for it, also known as life-time of an object. In C, an extent can have automatic, static extent, or dynamic extent.

Automatic Extent

An object with an automatic extent is created each time its declaration is encountered and is destroyed each time its block is exited.
For Example, a variable in the body of a loop is created and destroyed in each iteration.

Static Extent

A variable with a static extent is created when the program is loaded for execution and is destroyed when the execution stops. This is true no matter how many times the declaration is encountered during the execution.

Dynamic Extent

Dynamic extent is created by the program through malloc and its related functions.

Linkage

A large application program may be broken into several modules, with each module potentially written by a different written by a different programmer. Each module is a separate source file with its own objects.
We can define two types of **linkages**: internal and external

Internal Linkage

An object with an internal linkage is declared and visible in one module. Other modules cannot refer to this object.

External Linkage

An object with an external is declared in one module but is visible in all other modules that declare it with a special keyword, extern.

Storage Class Specifiers

There are four storage classes

- auto
- extern
- register
- static

Auto Variables

- A variable with an **auto** specification has the following storage characteristic:

Scope: block Extent: automatic Linkage: internal

- The default and the most commonly used storage class is auto.
- Memory for automatic variables is allocated when a block or function is entered. They are defined and are "local" to the block.
- When the block is exited, the system releases the memory that was allocated to the auto variables, and their values are lost.
- These are also referred with local variables

Declaration:

auto type variable_name;

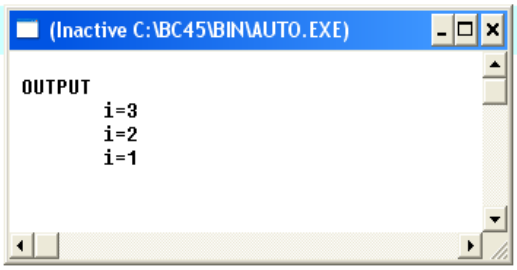
There's no point in using auto, as it's implicitly there anyway. By default all variables are created as automatic variables.

Initialization

An auto variable can be initialized where it is defined or left uninitialized. When auto variables are not initialized its value is garbage value.

// C Program to illustrate the auto storage class

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



In the above example, the compiler treats the three i's as total different variables, since they are defined in different blocks. Once the control comes out of the innermost block the variable i with value 3 is lost, and hence, i in the second printf () refers to i with value 2.Simillarly, when the control comes out of the next innermost block, the third printf () refers to i with value 1.

Register Variable

- register tells the compiler that the variables should be stored in high-speed memory registers if possible. This is done for efficiency.
- Only a few such registers are available, and can be assigned to frequently-accessed variables if execution time is a problem.

- If no register space is free, variables become auto instead. Keep registers in small local blocks which are reallocated quickly.
- A register variable address is not available to the user .this means we can't use the address operator and the indirection operator (pointer) with a register.
- A variable with **register** specification has the following storage characteristic:

Scope: block Extent: automatic/Register Linkage: internal

Declaration

register type variable_Name;

Initialization

A register variable can be initialized where it is defined or left uninitialized. When auto variables are not initialized its value is garbage value.

Example:

```
#include<stdio.h>
int main ()
{
   register double i=1;
   for (i=1; i<=1000; i++)
   {
                printf (" i=%d", i);

 }
}
```

register is freed on block exit

Static Variables

The static specifier has different usages depending on its scope.

1. Static Variables with Block Scope

- A static variable that is declared in a block scope, **static** defines the extent of the variable.
- The computer allocates storage only once. The linkage is internal, and not visible to other blocks.
- Also Know as static Local Variable.
- A variable with an **static block** specification has the following storage characteristic:

Scope: block Extent: static Linkage: internal

Declaration

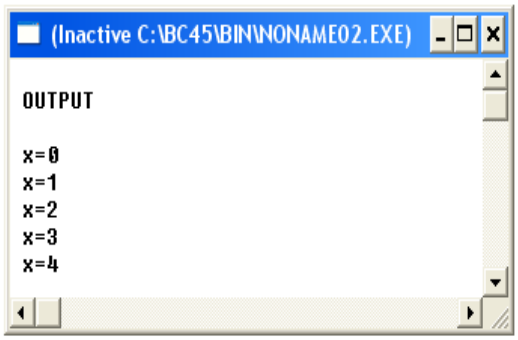
static type variable_Name;

Initialization

A static variable name can be initialized where it is defined, or it can be left uninitialized. If it is initialized, it is initialized only once. If not initialized, its value will be initialized to zero.

// C Program illustrating static storage class in block

```
#include<stdio.h>
void fun1();
int main()
{
    int i=1;
    printf("\n OUTPUT \n");
    for(i=1;i<=5;i++)
            fun1();
    getch();
    return 0;
}
void fun1()
{
     static int x=0;
     printf("\n x=%d",x++);
}
```



Note that in the above program variable i is an auto variable. The variable x, however, is a static variable .It is only initialized once although the declaration is encountered 5 times. I we do not initialize x to 0 because a static value needs an initialization value.

Note: The difference between a static local variable and a global variable is that the static local variable remains known only to the block in which it is declared. Global variable is visible to all the blocks in the program.

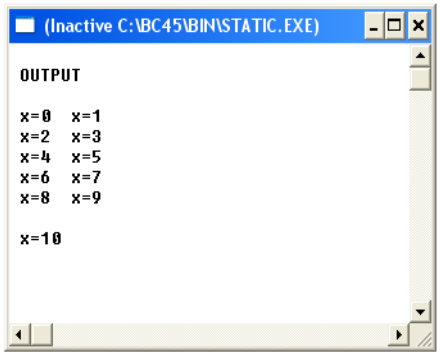
2. Static Variable with File Scope

- When the static specifier is used with a variable that has file (global) scope and we want keep its internal linkage, it is defined with specifier static.
- A variable with an **static file** specification has the following storage characteristic:

Scope: file Extent: static Linkage: internal

//C Program illustrating static specifier with file scope

```
#include<stdio.h>
void fun1();
static int x;
int main()
{
     int i=0;
     printf("\n OUTPUT \n\n");
     for(i=1;i<=5;i++)
      {
            printf(" x=%d ",x++);
            fun1();
      }
      printf("\n x=%d",x);
      return 0;
}
void fun1()
{
      printf(" x=%d\n",x++);
}
```



The declaration of the file scope must be in the global area of the file. If there is another declaration with the same identifier in the global area, we get an error message. In the above program x in declared in global area, it is visible for all blocks in the program. Here we can observe the control transfer between the function block.

External Variables

- External variables are used with separate compilations. It is common, on large projects, to decompose the project into many source files.
- The decomposed source files are compiled separately and linked together to form one unit.
- Within file variables outside functions have external storage class, even without the keyword **extern**.

- Global variables (defined outside functions) and all functions are of the storage class extern and storage is permanently assigned to them.
- Files can be compiled separately, even for one program. **extern** is used for global variables that are shared across code in several files.
- a variable declared with a storage class of extent has a file scope; the extent is, static, but linkage is external.

Scope: file Extent: static Linkage: external

Declaration

extern type variable_Name;

An external variable before must be declared before that is used by other source files. The above syntax is used to external variables
from other files.

Initialization

If the variable not initialized, then the first declaration seem by the linkage is considered the definition and all other declarations reference it. If an initializer is used with a global definition it is defining declaration.

```
// extern in multi-file projects
file1.c
        #include <stdio.h>
        int a =1, b = 2, c = 3;    /* external variables */
        int f (void);
        int main (void)
        {
          printf ("%3d\n", f( ));
          printf ("%3d%3d%3d\n", a, b, c);        print 4, 2, 3 ——
          return 0;
        }                                          a is global and changed by f

 file2.c
        int f (void)
        {
        extern int      a;      /* look for it elsewhere */
        int             b, c;
         a = b = c = 4;         ——  b and c are local and don't survive
        return (a + b + c);
        }                               return 12
```

Note: The difference between a normal static variable and a extern static variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

class	Scope	Extent	Linkage	Initial Value
auto	block	automatic	internal	indeterminate
register	block	automatic	internal	indeterminate
static(extern)	block	static	internal	Zero
static(linkage)	file	static	internal	Zero
extern	file	static	external	indeterminate

Table: Summary of Storage Classes