Keyword	Place of declaration	Lifetime	Scope	Initial value	Place of Storage	Linkage
auto	Inside function/ block	Function/ block	Function/ block	Gasbage	Memory (Stack)	None Variable
Register	"	"	"	"	Registers	None
Static (local)	"	Program	"	Zero	Memory (data area)	"
Static (global)	Outside or inside fun (declaration of external variable)	Program/ Function	Declaration to end of file		Memory (data area)	External
Extern	Outside function	Program	Definition to end of file	Zero	Memory (data area)	Internal

Storage Class Keyword :-

- While declaring the variable we can mention storage class keyword.
- By observing the storage class compiler decides four important properties to a variable those are

1. Default value
2. Memory allocation
3. Scope
4. Life.

These are four storage class keyword available in C, those are

1. Auto
2. Static
3. Extern
4. Register.

- In these four keyword all of them we can use while declaring the variable whereas static and extern we can use for function also.
- We can declare a variable in our program in two places :
  1) Above the main which is called as global variable.
  2) Within the function which is called as local variable.

If we declare a local variable without mentioning any storage class type, compiler considers them as auto variable.

- Every auto variable is a local variable but every local may not be auto.

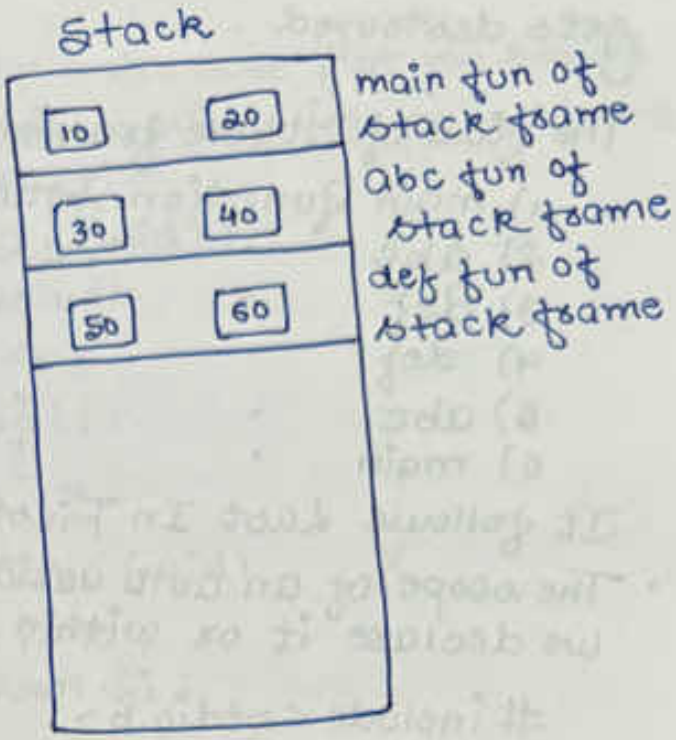
```
#include <stdio.h>
auto int m ;   — Error i.e, we can't use auto variable
void main()        as global variable
{ int i ;          — auto
  auto int j ;     — auto
  static int k ;   — static
}
```

Stack frame
auto variable
formal argument
return argument

- Default value of auto variable is a garbage or unknown value.

- Auto variable are stored in respected function stack frame.

```
#include <stdio.h>
void abc (void);
Void def (void);
Void main ()
{
  int i=10, j=20;
  abc ();
}
void abc (void)
{
  int k=30, l=40;
  def ();
}
void def (void)
{
  int m=50, n=60;
}
```



- Whenever our program is loaded into the RAM for execution operating system reserves 8 mb of stack for that particular program.
- The stack is divided into small frames depends on the function calls in our program.
  - In the above example - stat calls the main then main function stack frame gets created. In that stack frame i & j variable memory gets allocated.
  - Main is calling abc function, then abc function stack frame gets created. In that stack frame k & l variable memory gets created.
  - abc is calling def, then def function stack frame gets created. In that stack frame m & n variable memory gets created.
  - Once def function execution completes, when it goes back to abc the def fun stack frame gets destroyed.
  - After completion of abc when my function program execution goes back to main, abc function stack frame gets destroyed.

- After completion of main when my program execution goes back to main - stat, a main function stack frame gets destroyed.

The flow of stack frame creation and destroy as follows:

1) main function stack frame created
2) abc " " " "
3) def " " " "
4) def " " " destroyed
5) abc " " " "
6) main " " " "

It follows Last In First Out.

- The scope of an auto variable is within a function where we declare it or within a block where we declare it.

```
#include <stdio.h>
void main()
{ int i = 10;                          ← 10
  printf("1) i = %d\n", i);
    { int i = 20;                      ← 20
      printf("2) i = %d\n", i);
    }
  printf("3) i = %d\n", i);            ← 10
}
```

- Life of an auto variable starts when the function stack frame is created and life end when the function stack frame destroyed.
- While designing the function we should not return the address of auto variable which we declare inside that function.

```
int *abc(void);
main()
{ int *p;
  p = abc();
  . .
}
int *abc(void)
{ int i = 10;
  return &i;
}
```

Dangling Pointer :- If a pointer is holding life ended variable address.

Note :- While designing any function make sure that you should not return the address of the variable whose life going to end after completion of that function execution.

```
void abc (int *);
void main ()
{ int i = 10;
  abc (&i);
}
void abc (int *p)
{ *p = 20;
}
```
✓

```
int *abc (void);
void main ()
{ int *p;
  p = abc ();
  ---
}
int *abc (void)
{ int i = 10;
  return &i;
}
```
✗

Static Storage Class :-

- Static variable can be declared local as well global but auto should be local.
- The default value of static variable is 0.
- Static variables are stored in data segment.
- Uninitialized static variables are stored in BSS (Block started by symbols) or (uninitialized data segment), whereas initialized static variables are stored in initialized data section.

```
#include <stdio.h>
static int i = 10;
void main ()
{ printf ("i = %d \n", i);
}
```
↳ 10

```
cc -c s1.c
size s1.o
```

text	data	BSS	dec	hex	filename
94	4	0	98	62	s1.o

```
#include <stdio.h>
static int i = 10;
void main ()
{ static int i = 20;
  printf ("i = %d \n", i);
}
```
↳ 20

Note :- We can declare a global variable as well as a local variable with same name. In such case local is the higher priority.

- There is no directly method to excess the global variables, if local is there with the same name but in C++, there is an operator called scope resolution operator (::) with that we can excess the global even local is there with the same name.

- Scope :- If we declare a static variable within a function the scope is inside that function only.
  If we declare a static variable outside the function (globally) the scope is within that file.

- Life :- Life of a static variable begins when program is loaded into the RAM for execution and life ends when the program execution ends.

***

- For static variable re-initialization will not happen

```
#include <stdio.h>
void abc (void);
void main ()
{ abc ();
  abc ();
  abc ();
}
void abc (void)
{ static int i = 10;
  i++;
  printf ("i= %d \n", i);
}
```

↳ 11
12
13

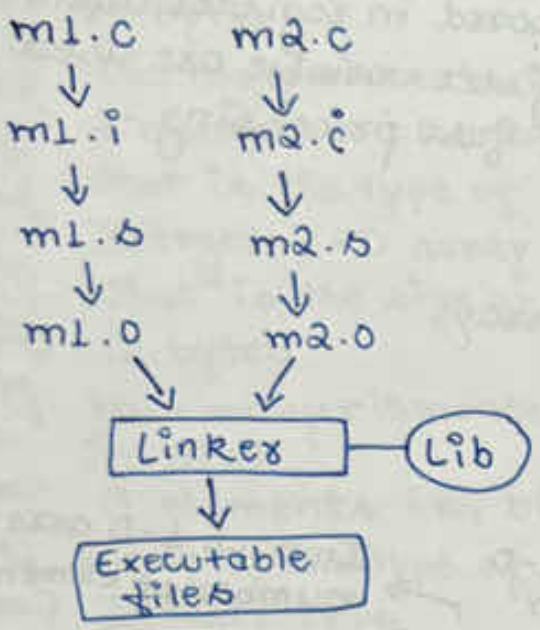
Q. How to access the global variable if the local variable is there with the same name.

```
#include <stdio.h>
static int i = 10;
int *abc (void);
void main ()
{
  static int i = 20;
  int *p;
  printf ("i= %d \n", i);
```

```
p = abc();
printf("*p = %d\n", *p);
}
int *abc(void)
{
int *abc
return (&i);
}
```

↳ i = 20
*p = 10

* Multiple File compilation :-



- By default function are non-static (extern). So, in one file if we declare one function, that function can be called from another file also.

Q. What is a static function ?
⇒ When we defining the function if we put static keyword before the definition that function becomes static function, so that function is visible only inside that file.
We cannot call that function from another file.

- Translator decides the scope (visibility) whereas linker decides who cannot access that.

Note :- Without declaring the function if we call the function translator generates warning.
Without declaring the variables if we call that variable translator generates error.

	Extern int i	int i	int i = 10
Translator	Declaration	Definition D + M (Memory allocated)	Initialization D + M + A
Linker	X	Weak variable or symbol	Strong Variable

External Linkage — int i = 10 ;
Internal Linkage — static int j = 20 ;
void main ()
{
External Linkage — int k = 30 ;
is not these (No Linkage) — static int l = 40 ;
}
void abc(void)
{
- - -
- - -
}

* auto & static variable are almost same.
- Local variable does not have any linkage.
- Register variable are stored in CPU register and if this register is busy it goes to stack.
- We cannot take input from keyboard in register variable Register variable are used for fast processing.

2-D Array

- 2D Array is a collection of 1'D's array
- 2D Array is a array of arrays
- 2D Array is also called as matrix

Q. How to declare a 2D Array

⇒ No. of 1-D array → In each 1-D array number of elements

datatype variablename [rows] [coloumn] ;

```
#include <stdio.h>
void main ()
{ int r, c ;
  int b[2][3] = { {10,20,30}, {40,50,60} } ;
  printf ("%d\n", sizeof (b)) ;
     "      "      "   (b[0])) ;
     "      "      "   (b[0][0])) ;
```

→ 24
12
4

10	20	30	40	50	60

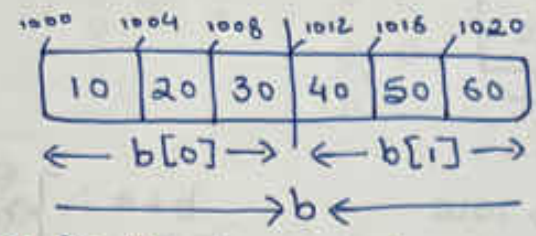
← b[0] → ← b[1] →

→ b ←

```
  r = sizeof (b) / sizeof (b[0]) ;
  c = sizeof (b[0]) / sizeof (b[0][0]) ;
  printf ("r = %d  c = %d\n", r, c) ;
}
```

r = 2
c = 3 .

```
int b[2][3] = {{10,20,30},{40,50,60}};
```

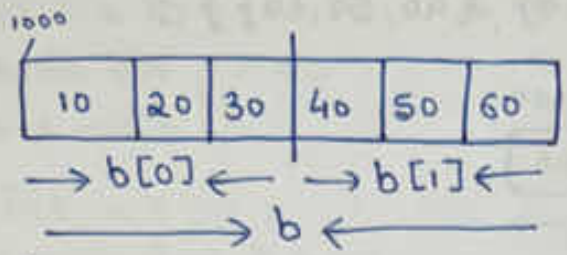


Q. What is the type of b.
⇒ Integer 2D array
Q. What is the type size of b.
⇒ 24 bytes
Q. How many elements are present in b.
⇒ 2 elements i.e., b[0], b[1]
Q. What is the type of b[0].
⇒ Integer 1-D array
Q. What is the size of b[0].
⇒ 12 bytes
Q. How many elements are present in b[0].
⇒ 3 elements i.e., b[0][0], b[0][1], b[0][2]
Q. What is the type of a b[0][0].
⇒ Integer type
Q. What is the sizeof b[0][0].
⇒ 4 bytes

```
for (i=0; i<r; i++)
{
  for (j=0; j<c; j++)
  printf ("%d", b[i][j]);
  printf ("\n");
}
```

```
int b[2][3] = {{10,20},{40,50}};
  // partial initialization
```

↳ 10 20 0
40 50 0



2D Array - b - 1000
1D — b[0] - 1000
int — b[0][0] - 10

&b - 1000
&b[0] - 1000
& b[0][0] - 1000

b+1 → 1012
b[0]+1 → 1004
b[0][0] → 10+1
+1 = 11

& b+1 = 1024
& b[0]+1 = 1012
& b[0][0]+1 = 1004

b++; } Error because array is a constant pointer
b[0]++;

b[0][0]++;
↓
No Error

* Difference between typedef and macro :-

Macro	Typedef
1) #define is a preprocessor directive	1) User defined datatype.
2) #define is useful for communication with processor	2) Typedef is useful for the communication with translator
3) It is used for data types, symbol replacement with another symbol.	3) Only for the data types.
4) Replacement takes place	4) No replacement.
5) #define IPTR int* IPTR P, q; int *P, q;	5) typedef int* INTPR P,q; INTPR P,q;
Syntax :- #define macroname microbody	Syntax :- Typedef oldname newname

* Typedef :- It is one of the user defined datatype which is used to provide another name to existing data type. With the help of this we can make understanding of variable or a data type.

Common Data Structure Operations and Array Sorting Algorithms

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	deletion	Access	Search	Insertion	Deletion	
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
SLL	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
DLL	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
B/BT	"	"	"	"	"	"	"	"	O(n)
Cartesian tree	N/A	"	"	"	"	"	"	"	"
Hash Table	"	O(1)	O(1)	O(1)	N/A	"	"	"	"

Algorithm	Best	Average	Worst	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Mergesort	"	"	O(n log(n))	O(n)
Heapsort	"	"	"	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	"
Insertion Sort	"	"	"	"
Selection Sort	O(n^2)	"	"	