5.1 Dynamic Memory Allocation:

- ✓ Dynamic Memory Allocation is not fixed, the no.of bytes reserved for the variable we can be changing during the execution of the program.
- ✓ It is a not fixed size of memory and not constant.
- ✓ Dynamic Memory Allocation in C Language is possible by 4 functions.

1. malloc( )
2. calloc( )
3. realloc( )
4. free( )

- ✓ These 4 functions are included in **stdlib.h**
- ✓ When we are using dynamic memory allocation, then memory allocated in heap memory.

1. malloc( ) : (Memory allocation)

- ✓ The malloc() function reserves a block of memory of specified size and pointers of type void *(generic pointer).

Prototype :

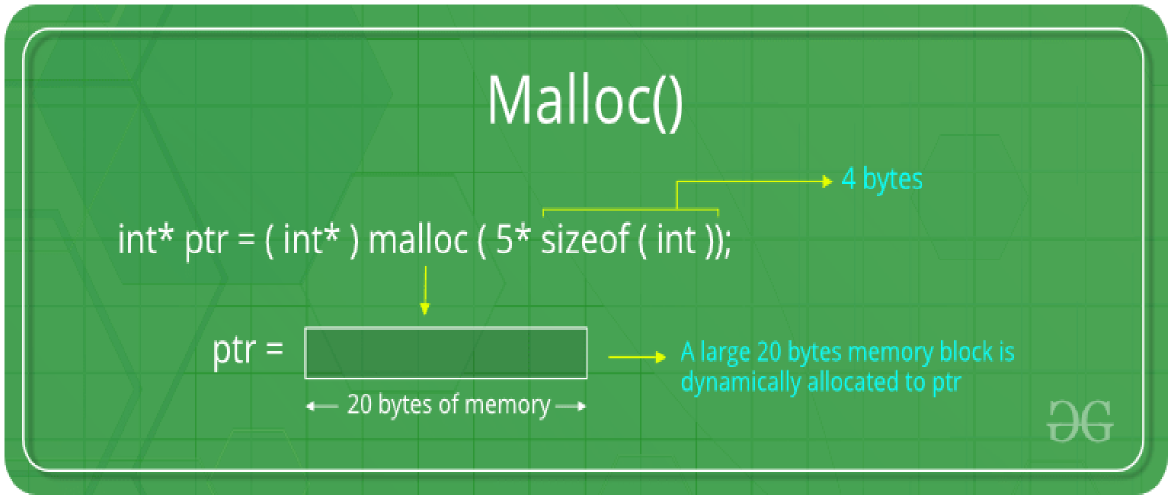
void * malloc(size of array *sizeof_datatype

Syntax:

ptr= (cast_type*) malloc (byte_size)

where ptr is a pointer, malloc( ) function returns the base address of the memory .

- ✓ The malloc( ) function successfully completed,it reserves memory, it returns base address.
- ✓ The malloc( ) function on failure, it reserves not reserve any memory, it returns null pointer.


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

Ex:

```
ptr=(int *)malloc(10*sizeof(int));
```

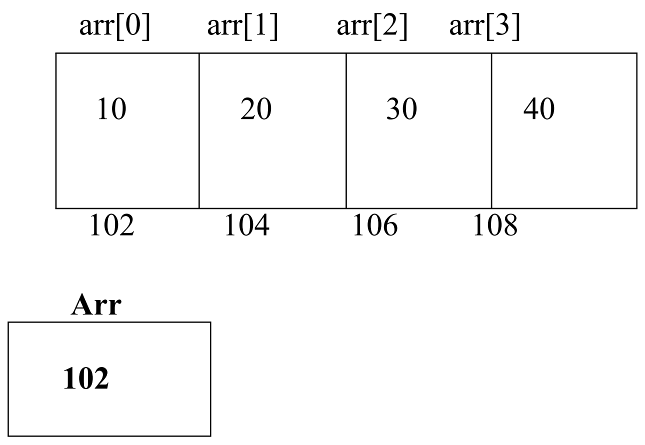
- ✓ malloc( ) function having single argument.
- ✓ malloc( ) function specially for structures. It is also possible for allocation of memory for arrays also. But this function specially dynamic memory allocation for structures.
- ✓ The default value is garbage value.

Example 1: (To Create arrays by using malloc( ))

```
#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i;

printf("Enter no.of element you want");

scanf("%d",&n);

int *arr;

arr=(int *)malloc(n*sizeof(int));

if(arr==NULL)

{

 printf("\n The NO Space in memory
allocation is failed");

exit(0);

}

for(i=0;i<n;i++)

{

       printf("\n Enter the Index %d of the array",i);

        scanf("%d",&arr[i]);

 }
```

OUTPUT:

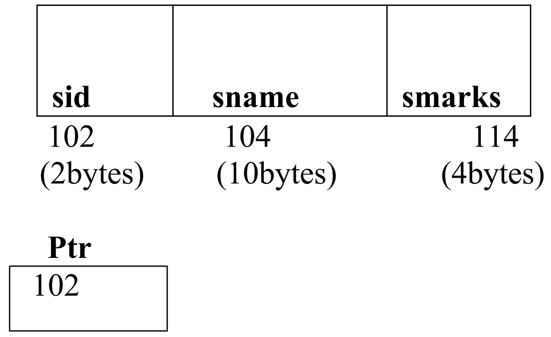
```
Enter no.of elements you want 4
Enter the index 0 of the array 10
Enter the index 1 of the array 20
Enter the index 2 of the array 30
Enter the index 3 of the array 40
The array contains
10 20 30 40
```



```
printf("\n The array contains\n");

for(i=0; i<n; i++)

{

 printf("%d\t",*(arr+i));

}

free(arr);

return 0;

}
```

Example 2: (To Create structure using malloc( ) function)

When we are using malloc() for structures memory allocated like below..



```
 #include<stdio.h>

#include<stdlib.h>

struct student

{

  int sid;

  char sname[10];

  float smarks;

};

int main()

{

struct student *ptr;

ptr=(struct student*)malloc(sizeof(struct student));

if(ptr==NULL)

{

   printf("No storage space");
```

OUTPUT:

```
Enter student ID: 1224
Enter student name: Ram
Enter student marks: 8.7

Student Details Are:
ID NO:1224
NAME: Ram
CGPA:8.7
```

```
}
else
{
                    printf("Enter student ID:");
                    scanf("%d",&ptr->sid);
                    printf("Enter student name:");
                    scanf(" %s",ptr->sname);
                    printf(" Enter student marks:");
                    scanf("%f",&ptr->smarks);
                    printf("Student Details Are:");
                    printf("\n ID NO:%d ",ptr->sid);
                    printf(" \n NAME: %s",ptr->sname);
                    printf("\n CGPA:%f",ptr->smarks);

}
}
```

Note: When we are using structures with pointer then we have to access the elements with arrow (->) operator.

2. calloc() (Contiguous memory allocation):

- ✓ calloc() is nothing but continuous memory allocation that is happed in arrays.so, calloc() function creates dynamic memory allocation for only arrays.

- ✓ This function allocates multiple blocks of request memory with specified size.

Prototype :

```
void *calloc(sizeof array,size of each element);
```

Syntax :

```
ptr= (cast_type *) calloc (no of elements,size of each elament);
```

where ptr is a pointer, calloc( ) function returns the base address of the memory .

- ✓ The calloc( ) function successfully completed,it reserves memory, it returns base address.
- ✓ The calloc( ) function on failure, it reserves not reserve any memory, it returns null pointer.

Ex:

ptr=(int *)calloc(10*sizeof(int));

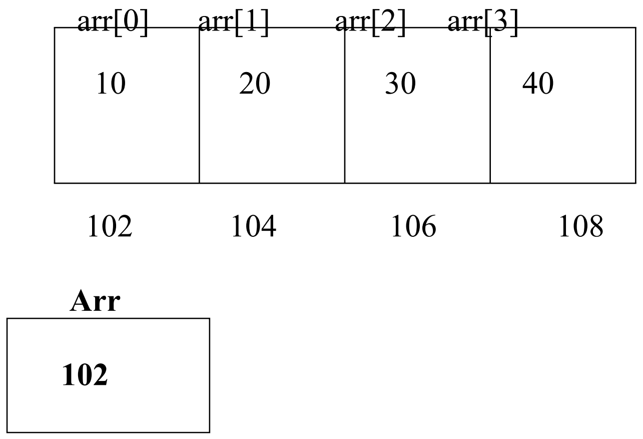
- ✓ In calloc( ), all elements are initialized with 0's if we not assigned any values.

Example:

```
#include<stdio.h>

#include<stdlib.h>

int main()

{

int n,i;

printf("Enter no.of element you want");

scanf("%d",&n);

int *arr;

arr=(int *)calloc(n,sizeof(int));

if(arr==NULL)

    {

            printf("\n The NO Space in memory
allocation is failed");

            exit(0);

    }

else

{
```

OUTPUT:

```
Enter no.of elements you want 4
Enter the index 0 of the array 10
Enter the index 1 of the array 20
Enter the index 2 of the array 30
Enter the index 3 of the array 40
The array contains
10 20 30 40
```



```
for(i=0;i<n;i++)

{

        printf("\n Enter the Index %d of the array",i);

        scanf("%d",&arr[i]);

 }

printf("\n The array contains\n");

for(i=0; i<n; i++)

{

printf("%d\t",*(arr+i));

}

}

free(arr);

return 0;

}
```

Difference between malloc() and calloc():

S.No	malloc()	calloc()
1	malloc() means memory allocation. this function reserves a block of memory of specified size. Ex : structures by using malloc() specially allocating memory for structures. It is also possible to allocate memory for arrays also.	calloc() means contiguous memory allocation. Continuous memory allocation happen at arrays. this function reserves multiple blocks block of memory of specified size. Ex: arrays by using calloc() specially allocating memory for arrays only because continues memory blocks occurs in array.
2	In this, default value garbage value if we are not initialized any value,	In this, default value zero. If we Not any values, then the compiler initialized with zero to all array elements.

3	malloc() function takes single argument Ex: (int *)malloc(n*sizeof(int))	calloc() function takes two arguments Ex: (int *)malloc(n,sizeof(int))
4	malloc() is faster than calloc().	calloc() is slow compare to malloc beacauseof the extra step of initializing the allocated memory by zero..

3.realloc( ): (Memory reallocation)

- ✓ Memory reallocation is nothing but at the time memory allocated by using calloc() or malloc( ) might be sufficient or in excess, that means to increase or decrease the memory.
- ✓ We can always use realloc( ) function to change the memory size already allocated by calloc( ) and malloc( ). This process is called reallocation of memory.
- ✓ With realloc(), you can allocate more bytes of memory without losing your data.

Syntax:

```
ptr = realloc(ptr,newsize)
```

Prototype:

```
void* realloc(void *ptr,size_t size);
```

Ex:

```
ptr = (int*)realloc(ptr,10)
```

- ✓ realloc( ) returns base adreess of the array or structure after decrease or increase memory.

Example:

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

int main()

{

    char *str;
```

```
 str=(char*)calloc(5,sizeof(char));
 if(str==NULL)
 {
            printf("memory could not be allocated");
            exit(1);
 }
 strcpy(str,"Hi");
printf("Before realloc():\n");
printf("\n String=%s",str);
str=(char*)realloc(str,10);
if(str==NULL)
 {
 printf("memory could not be allocated");
exit(1);
 }
 printf("\n String size is modified");
 printf("\n Checking the previous data
                      exists or not");
 printf("\n String=%s",str);
printf("\n Append new string to previous
                      String");
 strcat(str," students");
 printf("\n String=%s",str);
 free(str);
 return 0;    }
```

OUTPUT:

```
Before realloc():
String=Hi
String size is modified
Checking the previous data exists or not
String=Hi
Append  new string to     previous String
String=Hi students
```

Explanation :

1) ptr=(int*)calloc(5,sizeof(char))

h	**i**	**\0**	**\0**	**\0**

String =Hi

2) ptr=(int*)realloc(ptr,10)

H	**i**		**s**	**t**	**u**	**d**	**e**	**n**	**t**	**s**	**\0**

String =Hi students

4.free( ):

- ✓ Releasing the used space or de-allocate the memory
- ✓ When a variable is allocated space, then the memory used by that variable is automatically released by the system.
- ✓ But when we dynamically allocate memory then it is our responsibility to release space when it is not required.
- ✓ This is more important when the storage space is limited.

Syntax : free(ptr)

- ✓ When ptr is a pointer that has been created by using malloc() or calloc(),where memory is de-allocated using the free( ).

Storage classes

The storage classes based on four parameters they are

1. Storage
2. Initial value
3. Scope (Scope of variable)
4. Life time of variable

1. Storage : (Where the variable would be stored)

From C compiler's point of view, a variable name identifies some physical location within the computer where the value of the variable is stored. There are basically two kinds of locations in a computer where such a value may be kept—

1. Memory **(RAM)**
2. CPU registers.

2. Initial value :

It specifies whether the variable will be automatically initialized to zero or unpredictable value(Garbage value), if initial value is not specifically assigned (i.e the default value).

3. Scope of the Variables :

- ✓ What is the scope of the variable; i.e. in which functions the value of the variable would be available.
- ✓ All variables have a defined scope, by scope we mean the **accessibility** and **visibility** of variables at different point in the program.
- ✓ There are four types of scope:
  a) Block Scope.
  b) Function / Method Scope.
  c) Program Scope.
  d) File Scope.

a) Block Scope:

- ✓ A block is a group of statements enclosed with opening and closing curly brackets {}.
- ✓ If a variable declared within the block then we access that variable within that particular block only we can't accesses that variable outside the block, such variable are called block scope variables. It is also known as Local variable i.e declare a variable inside the function is called local variable.
- ✓ The scope the variable access particular block where the variable declared.

Example :

```
#include<stdio.h>
void main()
{
   {
      int a=10;
      printf("the value of A is :%d",a); → here prints a value is 10;
   }
  printf("the value of A is :%d",a);  → here prints an error that is a is not declared in this function
}                                       because a is defined inside of another block, we can't
                                        Access that variable outside the block.
```

Example 2:

```
#include<stdio.h>
void main()
{
```

```
            int a=24;
            int i=1;
            printf("the value of outside the block is %d",a);  → prints 24
            while(i<4)
             {
                  int a=i;
                  printf("the value of inside the block is \t %d",a); → prints 1 2 3
                  i++;
             }
        printf("the value of outside the block is %d",a);    → prints 24
    }
```

Note: In the above program we declared and initialized an integer variable ‘a’ in main() function and then re-declared and re initialized in a while loop, then the variable ‘a’ will be considered as two different variables and occupy different memory slots.

b) Function / Method Scope:

- ✓ A variable declare inside the function, We can access a variable starting to the end of a function is known as function scope.
- ✓ We can also access a variable inside blocks or loops but you should not declare a variable with same name then only we can access.
- ✓ This variable also known as a local variable.

Example 1: with function scope and block scope variables.

```
#include<stdio.h>
void main()
{
   int  a; //function scope
   {
     int a=10; // block scope
     printf("%d",a); → prints 10
   }
 printf("%d",a); → prints garbage value
 test( );
}

void test( )
{
 printf("%d",a); → Error the function
                   scope variable we can't
}                  access outside the
                   function, here the error
                   a is not declared.
```

Note : In this program, function scope and block scope variable both are different variable with same name.

Example 2: function scope without block scope variables.

```
#include<stdio.h>
void main()
{
   int  a; //function scope
   {
      a=10;
     printf("%d",a); → prints 10
   }
 printf("\t%d",a); → prints 10

}
```

OUTPUT: 10 10

Note: In this program we are not declared block scope variable then we access a function scope variable inside the block because the block is declared inside the function. If we declared a variable with same name of function scope variable inside the block then we can't access the function scope variable. Block scope variable is overrides the function scope variable.

c) Program Scope:

- ✓ A variable declare outside the function ,We can access a variable in entire program is known as program scope.
- ✓ It is also known as global variable. If we declare a variable outside the function is known as global variable.
- ✓ Global variable are not limited to a particular function so they exists even when a function calls another functions. These variables can be used from every function in the program.
- ✓ The global variables are declared outside the main() function.
- ✓ If we have a variable declared in a function that as same name as global variable, then the function will use the local variable declared within it and ignore the global variable.

Example 1:
Program scope with function and block scope variables

```
#include<stdio.h>
int  a=10; // program scope
void test( );
void main( )
{
   int  a=20;  //function scope
    {
      int a=30; // Block scope
      printf("\n value of A inside the block
                    scope :%d",a);
    }
  printf(" \nvalue of A inside the function
                    scope :%d",a);
test( );
}

void test( )
{
 printf("\nvalue of A in other functions:%d",
                              a);
}
```

```
OUTPUT:
value of A inside the block scope: 30
value of A inside the function scope: 20
value of  A inside in other functions: 10
```

Note:

- ✓ If we not initialized global variable values then compiler places the default integer value '0'.
- ✓ In above program the function and block scope 'a' variables overrides global variable 'a'.

Example 2:
Program scope without function and block scope variables.

```
#include<stdio.h>
int  a=10; // program scope
void test( );
void main( )
{
     a=20;
     {
       a=30;
       printf(" \nvalue of A inside the block
                    scope :%d",a);
     }
  printf(" \nvalue of A inside the function
                    scope :%d",a);
test( );
}

void test( )
{
 printf("\nvalue of A in other functions:%d",
                              a);
}
```

```
OUTPUT:
value of A inside the block scope: 30
value of A inside the function scope: 30
value of  A inside in other functions: 30
```

Note:

- ✓ In the above program we declare only global variable, but initialized value to global variable in function and block scopes.

d) File Scope :

When a global variable is accessible until the end of the file, the variable said to have file scope. Declare variable with the static keyword.

Ex : static int a=10;

4. Life time of variable: (how long would the variable exist)

- ✓ The life time of any variable is the time for which the particular variable outlives in memory during the running of the program.

4.3.1 Types of Storage Classes

There are four storage classes in C:

1. auto
2. register
3. static
4. external

Automatic Variables

These are declared inside a function in which they are to be utilized. These are declared using a keyword *auto.*

Example:

auto int number;

- ✓ These are created when the function is called and destroyed automatically when the function is exited.
- ✓ These variables are private (local) to the function in which they are declared.
- ✓ Variables declared inside a function without storage class specification is, by default, an automatic variable. i.e int a =10; and auto int a=10; both are same.

Properties of automatic variable are as under:

Storage	Memory.
Initial value	An unpredictable value, which is often called a *garbage value.*
Scope	Local to the block in which the variable is defined
Life	Till the control remains within the block in which the variable is defined.

Example 1:

```
#include<stdio.h>
void main( )
{
  auto  int a=10;
    {
      auto int a;
      printf("the value inside block:%d\n",a);
    }
  printf("the value outside block:%d\n",a);
}
```

OUTPUT:

the value inside block: **Garbage value**
the value outside block: **10**

Example 2:

```
#include<stdio.h>
int main( )
{
auto int i=1;
{
auto int i=2;
{
autoint i=3;
printf ( "\t%d ", i ) ;
}
printf ( "\t%d ", i ) ;
}
printf ( "t%d", i ) ;
return 0;
}
```

OUTPUT:

3 2 1

Register variables

These variables are stored in one of the machine's register and are declared using keyword *register*.

Example

register int count;

- ✓ Since register access are much faster than a memory access keeping frequently accessed variables in the register lead to faster execution of program.
- ✓ Use register storage class for only those variables that are being used very often in a program (loop counters).

Properties of external variable are as under:

Storage	CPU registers
Initial value	Garbage value
Scope	Local to the block in which the variable is defined.
Life	Till the control remains within the block in which the variable is defined

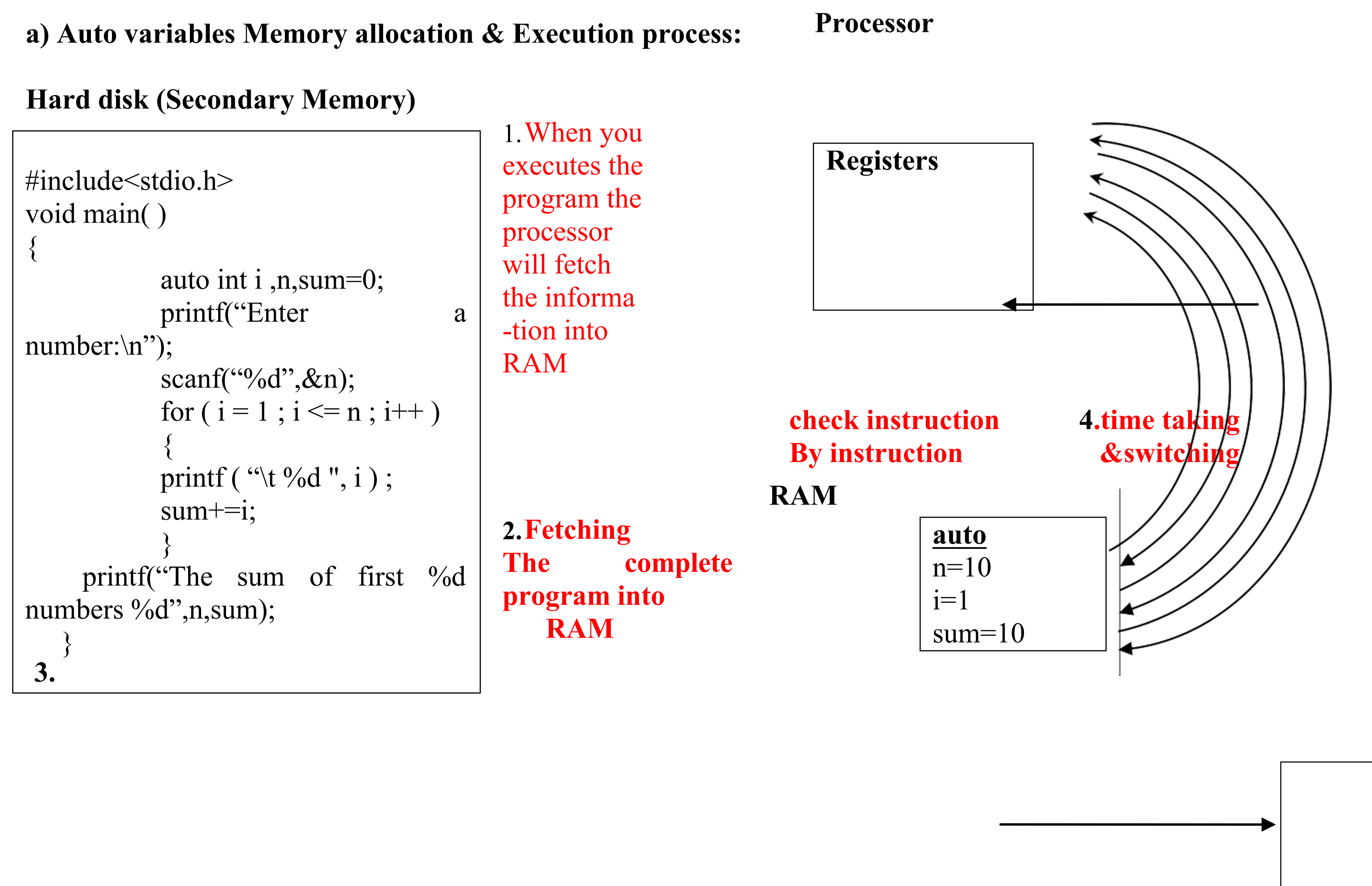
Example :

```
#include<stdio.h>
void main( )
{
            auto int i ,n,sum=0;
            printf("Enter a number:");
            scanf("%d",&n);
            for ( i = 1 ; i <= n ; i++ )
            {
            printf ( "\t %d ", i ) ;
            sum+=i;
            }
        printf("The sum of first %d numbers %d",n,sum);
   }
```

```
OUTPUT:
Enter a number: 10
1 2 3 4 5 6 7 8 9 10
The sum of first 10 numbers 55
```

Memory allocation and Execution process of auto and register variables:

a) Auto variables Memory allocation & Execution process:

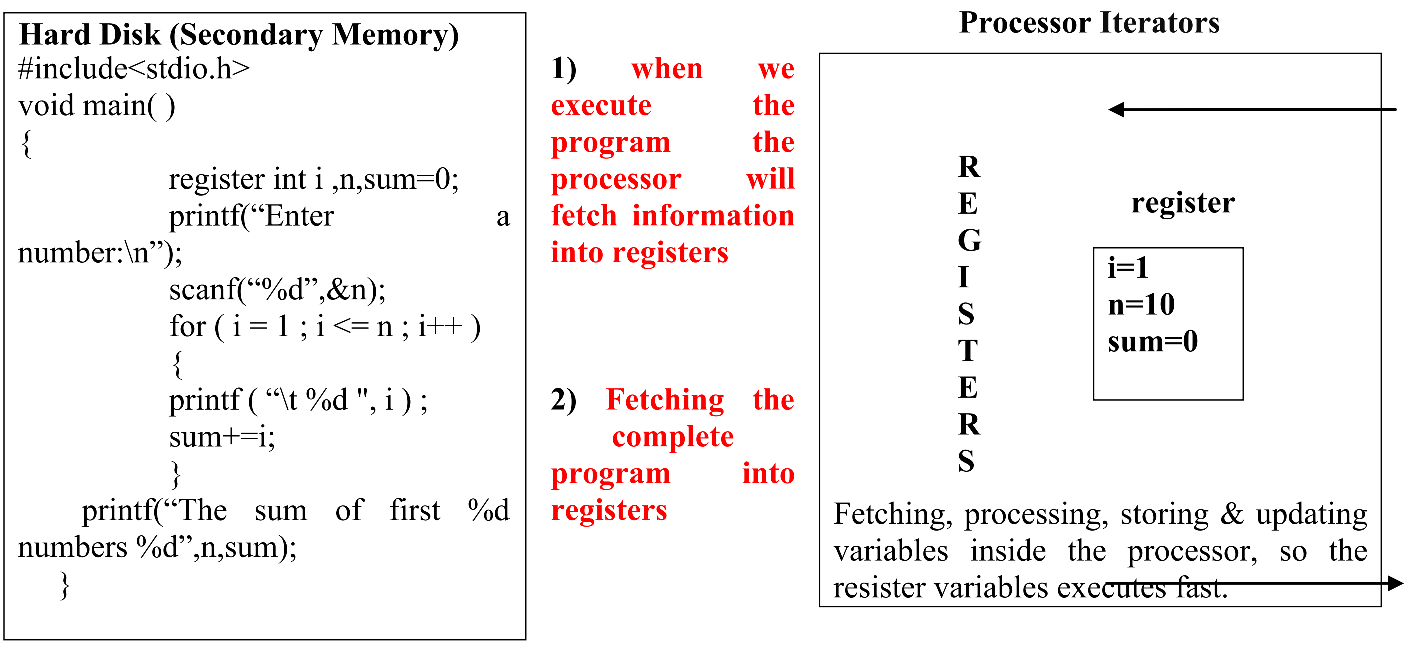

Processor
Hard disk (Secondary Memory)
1. When you executes the program the processor will fetch the informa -tion into RAM
Registers
check instruction By instruction
4.time taking &switching
RAM
2.Fetching The complete program into RAM
auto
n=10
i=1
sum=10
3.

```
#include<stdio.h>
void main( )
{
            auto int i ,n,sum=0;
            printf("Enter a number:\n");
            scanf("%d",&n);
            for ( i = 1 ; i <= n ; i++ )
            {
            printf ( "\t %d ", i ) ;
            sum+=i;
            }
     printf("The sum of first %d numbers %d",n,sum);
   }
```

In the above figure we observe execution process based 4 steps they are:

1. When we execute executes the program processor will fetch the information into RAM.
2. Processor fetch the complete program to RAM. The auto variable memory allocation in RAM only.
3. The processor collect information instruction by instruction from the RAM. The processor executes single instruction at a time.
4. For fetching, processing, storing and updating variables and control information to one location to another(RAM to processor and processor to RAM).The processor will performs fetching and processing and updating the data instruction by instruction and returns results to RAM. In this every time processor takes instruction from RAM and executes and update the variable and returns value to RAM upto number of times. This process will occur in looping concepts. the drawback of auto variable is switching and time taking process to execute the variable repeatedly. To overcame this we go for register variables.

b) <u>Register variable Memory allocation & Execution process:</u>



Hard Disk (Secondary Memory)

```
#include<stdio.h>
void main( )
{
        register int i ,n,sum=0;
        printf("Enter a number:\n");
        scanf("%d",&n);
        for ( i = 1 ; i <= n ; i++ )
        {
        printf ( "\t %d ", i ) ;
        sum+=i;
        }
    printf("The sum of first %d numbers %d",n,sum);
  }
```

1) when we execute the program the processor will fetch information into registers

2) Fetching the complete program into registers

Fetching, processing, storing & updating variables inside the processor, so the resister variables executes fast.

In the above figure we observe execution process based following steps :

1. When we execute the program the processor will fetch the complete program into register.
2. Processor will fetch complete program into registers.
3. Fetching , processing, updating & storing the data inside the processor . no need to send and receive information from other places, so the register variable executes faster than other variables.

Question: In this concept, instead of all the variables (auto variable) why can't we declare register variable? Directly we can remove the auto storage class , if only maintain register storage class, it better to every program executes faster ?

Answer : Here the problem is register size is very less it may 8KB to 16KB, this much memory is always executes instruction not to store the variable data. So the register storage class always prefer to executes and storing looping programs for faster execution.

Static variables

The value of static variables persists until the end of the program. It is declared using the *static* keyword.

Example: static int x;

- ✓ Static variables are initialized only once, when the program is compiled.
- ✓ Use *static* storage class only if you want the value of a variable to persist between different function calls.

Properties of static variable are as under:

Storage	Memory
Initial value	Zero
Scope	Local to the block in which the variable is defined
Life time	Value of the variable persists b/w different function calls. Ending of program

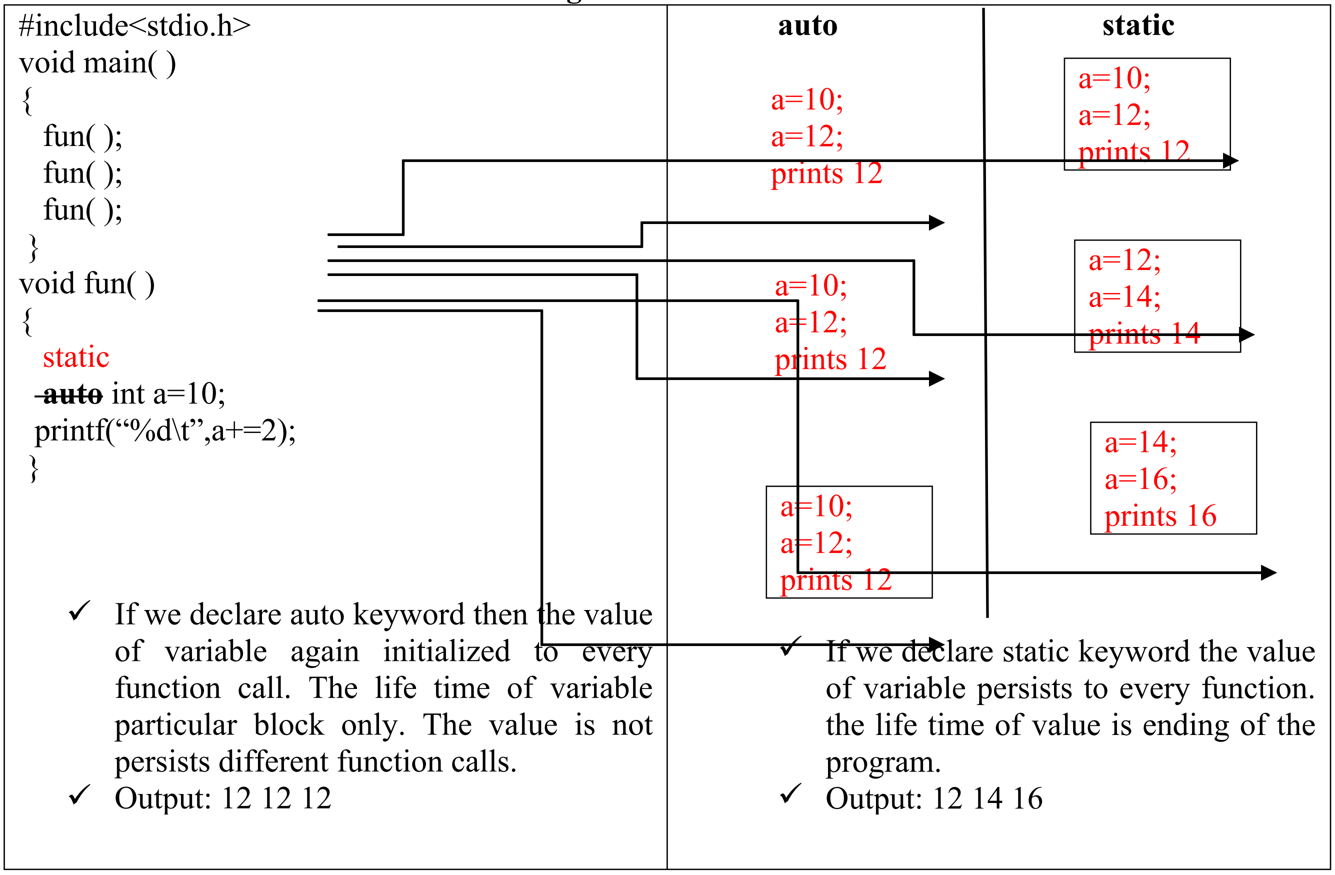
Example

```
#include<stdio.h>
void increment();
int main( )
{
            increment();
            increment();
            increment();
            return 0;
}
void increment()
{
            static int i ;
            printf ( "%d\t", i ) ;
            i = i + 1 ;
}
```

OUTPUT:

```
0           1           2
```

Difference between auto and static storage classes:



```
#include<stdio.h>
void main( )
{
  fun( );
  fun( );
  fun( );
 }
void fun( )
{
  static
 ~~auto~~ int a=10;
 printf("%d\t",a+=2);
 }
```

- ✓ If we declare auto keyword then the value of variable again initialized to every function call. The life time of variable particular block only. The value is not persists different function calls.
- ✓ Output: 12 12 12

- ✓ If we declare static keyword the value of variable persists to every function. the life time of value is ending of the program.
- ✓ Output: 12 14 16

External Variables

- ✓ The extern keyword is used with a variable to inform the compiler that variable is declared somewhere else.
- ✓ The extern storage class is used to give a reference of a global variable that is visible to all program files.
- ✓ It is similar to global variable.
- ✓ These variables are active and alive throughout the entire program.
- ✓ The keyword *extern* used to declare these variables.
- ✓ Unlike local variables they are accessed by any function in the program.
- ✓ In case local and global variable have the same name, the local variable will have precedence over the global one.

Properties of external variable are as under:

Storage	Memory
Initial value	Zero
Scope	Global and file scope.
Life time	Until the ending of the program's execution.

Example1

```
#include<stdio.h>
void main( )
{
 int x=24;
 extern int y;
 printf("x=%d\n",x);
 printf("y=%d\n",y);
}
y=10;
```

OUTPUT:

```
x=24
y=10
```

Example2

file1.c

```
#include<stdio.h>
#include"file2.c"
void main( )
{
 extern int x;
 extern int y;
 printf("x=%d\n",x);
 printf("y=%d\n",y);
}
```

OUTPUT:

```
x=24
y=1224
```

file2.c

```
#include<stdio.h>
x=24;
int y =1224;
```

Example3

File3.c

```
#include<stdio.h>
#include"file4.c"
void main( )
{
  extern int a;
  printf("%d\n",a);
  fun( );
  printf("%d\n",a);
}
```

OUTPUT:

```
7
8
8
```

File4.c

```
#include<stdio.h>
int a=7;
void fun( )
{
a++;
printf("%d\n",a);
}
```