UNIT-I

Storage Classes-

A storage class defines how variables and functions are stored in memory.

Variable declaration is the important step of any program. To store different values we need a variable. While declaring the variables we have to specify its 'data-types' as well as 'storage class' i.e. all the variables always have some default storage class. When a variable is defined it gets stored at particular location in computer.

There are mainly two types of locations where values can be stored:-

1) memory 2) CPU registers.

The storage class determines these locations.

These storage classes tells us-

i) Where the variable would be stored.

ii) What will be the default initial value of the variable if not specified.

iii) Scope of the variable i.e. in which function the variable is available.

iv) Life of of a variable.

There are four storage classes in c

a) Automic storage class.

b) Register storage class

c) Static storage class

d) External starage class

*** Automic storage class-**

This is the default storage class for all the variables declared inside a function or a block. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

It is the default storage class for any variable if not specified.

Its characterstics are:

Storage	: Memory
Default initial value	: A garbage value or unpredictable value.
Scope	: Is accessible to the block only in which it is defined.
Life	: Remain in existence till the control is within the block.

The variable should be declared using auto keyword. It gets stored in memory.

Example: Program to illustrate auto storage class.

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

```
auto int i,j;
printf(" \n %d %d ", i,j);
}
```

The above program will print any two values

2765 1211

Where 2765, 1211 are the garbage value of **i** and **j**. these variable are not accessible by any other functions in the program. Its scope and life is restricted for the procedure in which it is defined.

*** register storage class-**

This storage class is used to declare static variables that have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

Registrer storage class is having following features.

Storage	: CPU register
Default initial value	: Garbage value
Scope	: Local to the block in which it is defined.
Life	: Till the control remains within the block.

Values stored in the CPU register can be accessed more faster than the values stored in memory. When any variables is occurring multiple times in the program, its storage class should be register i.e. loop counter variables can be stored in the register as its values changes many times in the program. Only the thing to remember is that CPU registers are limited. If the variable doesn't get stored into register because of unavailability of space then the variables will be considered as auto. We can not use register storage class for all types of variables.

Example: program that demonstrate register storage class

```
#include<stdio.h>
Void main()
{
register i;
for(i=1;i<=20;i++)
{
printf("\n %d");
}
}
```

Here variable **i** is stored in register so it gets accessed faster in the loop

*** Static Storage Class-**

This storage class is used to declare static variables that have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

Following are the features of static storage

Storage : Memory

Default initial value : Zero

Scope : local the block in which it is defined

Life : value of the variable remains static between different function call.

The value of static variable remains alive to the whole program. That is the variable gets initialized only once.

Example: program that illustrates static storage class

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

Here the variable i is defined as static in the function Increment() and the function is called three times in The main function. The value of i will be printed as:

1

2

3

The variable gets initialized only once. For other function calls, the value remains as it is. So the output is in incremented form. Static variable are also local to the block in which it is defined. If the variable is defined as auto every time the value of I will be printed as

1

1

1

Because the variable gets initialized every time the function is called.

*** External storage class**

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.

Also, a normal global variable can be made extern as well by placing the 'extern' keyword before its declaration/definition in any function/block.

The features of the external storage class

Storage : memory

Default initial value : zero

Scope : global

Life : as long as the global.

They are declared outside all the functions. They can be accessible by all the functions in the program.

Example : program for extern storage class

```
#include<stdio.h>
void Increment()
void decrement()
extern int i =1;
void main()
{
printf("\n %d", i);
increment();
decrement();
i=i+1;
printf("\n %d", i);
}
void increment()
{
i=i+1;
printf("\n %d", i):
}
void increment()
{
i=i-1;
printf("\n %d", i):
}
```

The output of the program is

```
1   //initial value of i
2   //changed by increment() function
1   // changed by decrement() function
0   // again changed in main() function
```

This shows that the variable accessible by all the functions in the program and its value also remains as it is. Initial value of I is 1 it gets incremented by the increment() function i.e 2, then it will be decremented by decrement() function i.e again 1 and finally it gets decremented by the main() function, so the final value of I is 0.

String-

In C language array of characters are called string.

A string is a collection of characters/ group of characters that ends with a null character.

String constant- sequence of 0 or more than characters enclosed between double quotes

"S", "abc"," ", " printf("hello word")

Each character of string occupies 1 byte of memory.

We have char, int, float and other data types but no 'string' data type in c.

String is not supported data type in c but it is a very useful concept used to model real entities like name, city, etc.

A string is an array of characters that ends with a null character.

Ex: India

The null character is represented by '\0'.

The ASCII value of null character is zero.

Strings variable- the array of character type is known as string variable. For ex. Char a[10].

Declaration of a String

Strings can be declared like a one dimensional array.

Syntax:

char string_name[size];

Example:

char name[30];

char dept[20];

String Initialization

The string can be initialized as follows:

char dept[10] = "CSE";

OR

char dept[] = {'a', 'b', 'c','/0'};

In the above example, „\0" is a null character and specifies end of the string. Here string is assigned character by character.

Example

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

```
char name[10];
int age;
printf("Enter your first name and age: \n");
scanf("%s %d", &name, &age);
printf("You entered: %s %d", name ,age);
getch();
 }
```

Output:

```
 Enter your first name and age:
Swati 21
```

Built in string manipulation function:

The C string functions are built-in functions that can be used for various operations and manipulations on strings. These string functions make it easier to perform tasks such as string copy, concatenation, comparison, length, etc. The **<string.h>** header file contains these string functions.

i) strlen (string exp) **:** returns the length of the given string.
ii) strcpy (string1, string2) **:** copies the string2 into string1
iii) strcmp (string1, string2) **:** compare two strings & returns the integer difference.
iv) strrev(string) **:** reverse the given string provided as arguments
v) strcat(string1, string2) **:** concatenate (append) string 2 with string 1
vi) strchr(s,c) **:** Find the first occurrence of a character in a string.
vii) **strlwr() - toupper** **:** converts a single character into uppercase
viii) **strupr() - tolower** **:** converts a single character into lowercase
ix) strset() **:** Modifying the original string directly in memory.

examples:

```
=#include <stdio.h>
#include <string.h>
int main() {
char s[] = "Department of computer science";              // Finding and printing length of string s
printf("%lu", strlen(s));
return 0;
}
```

Output:

30

array of character-

- A Character array is a derived data type in C that is used to store a collection of characters or strings.

- A char data type takes 1 byte of memory, so a character array has the memory of the number of elements in the array. (1* number_of_elements_in_array).
- Each character in a character array has an index that shows the position of the character in the string.
- The first character will be indexed 0 and the successive characters are indexed 1,2,3 etc...
- The null character \0 is used to find the end of characters in the array and is always stored in the index after the last character or in the last index.

Initialization of character array-

Character array can be initialized in following ways:

Char name[10]; //declares a character array

Or

Char name [5]={'A','R','R','A','Y'}; //initialize the elements with five character

Or

Char name[5]={'a','b','c'}; // stores three characters to the array elements and remaining elements contains blank

Or

Char name[]={'S','M','I','L','E'}; // takes the size of array according to the characters stored.

Example- char name[8]={'W','E','L','C','O','M','E'};

Declares name as character array(string) that hold maximum of 7 character string"WELCOME". Each character of the string is treated as an element of array name and stored in the memory as follows.

W	E	L	C	O	M	E	/0

Name end of the string

Example: program to print 5 characters stored in memory

```
#include<stdio.h>
#include<conio.h>
Void main()
{
char str[5]={'A','R','R','A','Y'};
int i;
clrscr();
printf("\n the array is"]);
for(i=0;i<5;i++)
{
printf("\n %c",str[i]);
```

```
getch();
}
```

Though **str** is a character array, its element are referred by integer value so **i** should be of integer type. **%c** operator takes a single character. The complete string(array) can be printed using **%s** operator. It can also be used with scanf() to accept a string **%s** considers the whole string at a time as:

Example: program to accept a string in array with %s operator using scanf() function.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[25];
clrscr();
printf("enter a string");
scanf("%s", &str)
printf("\n the array is : %s",str);
getch();
}
```

Here for loop is not required and we have to specify the array name only in the printf() and scanf() statement. Another way to access the array elements is using a pointer.

The gets() and puts() function

In above example scanf() is used to accept a string. If the string is entered as " Happy Diwali" then only the word "Happy" will gets stored in the array and it will not terminate the string by \0. i.e. scanf() is not capable of receiving muiti-word strings, sentences or space between characters. To overcome this limitations **gets() function** can be used. Get() function can accept blank spaces and it stores it into the array. To this function we have to pass the array name. it also automatically strores '**\0'** to end the string.

Syntax- gets (arrayname);
Example: gets (name); // where name is character aaray.
Similarly puts() function works. It prints the whole string at a time on screen.
Syntax: puts(arrayname);
Example: puts(name)
Following program illustrates the use of gets() and puts() function.

```
#include<stdio.h>
void main()
{
char str[25];
int i=0;
```

```
printf("\n enter  a string :");
gets(str);
while(str[i]!='\0')
{
if(str[i]>=97 &&str[i]<123)
{
str[i]=str[i]-32;
}
i++;
}
printf("\n string is uppercase :");
puts(str);
getch();
}
```

Strings & pointers-

String pointers are pointers that point to the first character of a string. They are powerful tools for string manipulation, allowing programmers to modify strings without copying them entirely.

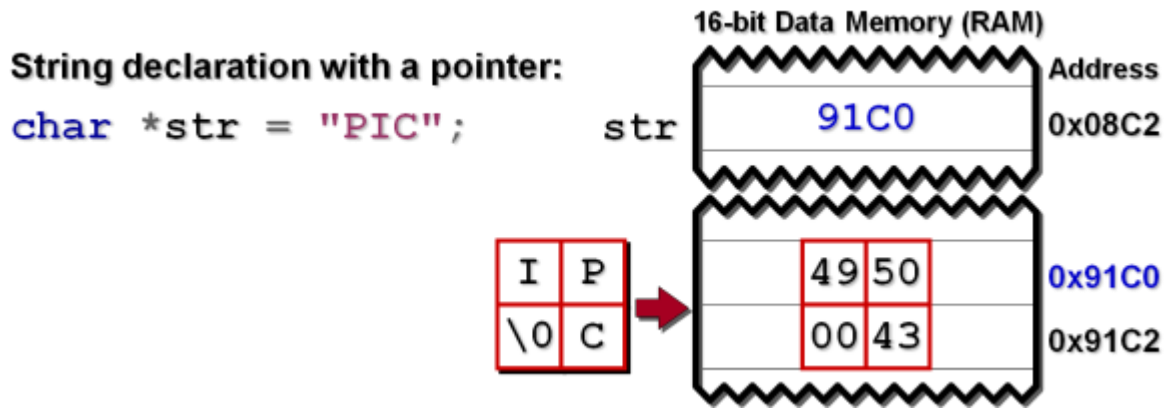
Declaring Pointer Variables for Strings

To declare a pointer to a string, you can use the syntax char *ptr = "hello";. This statement creates a pointer ptr that points to the first character of the string "hello".

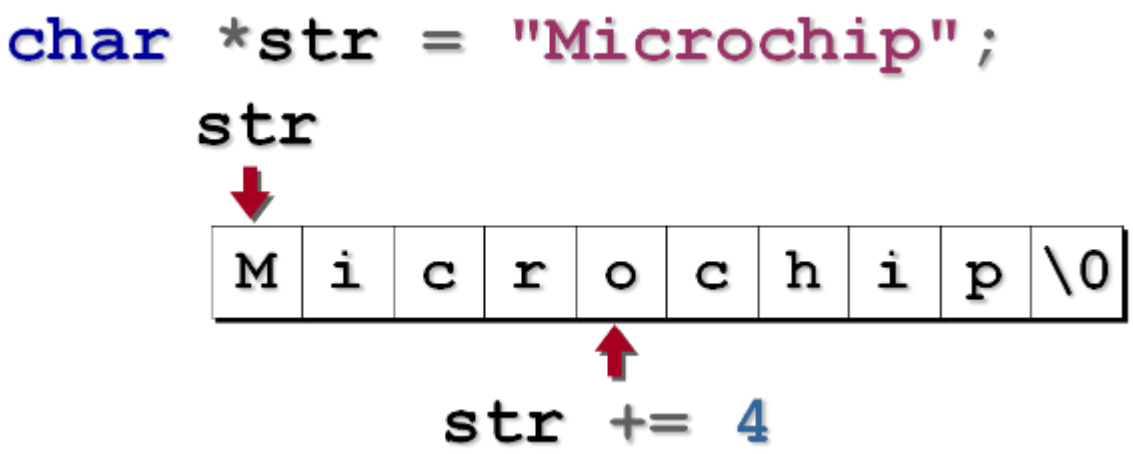
Accessing Strings with Pointers

Accessing strings through pointers involves understanding how pointers can be used to traverse and manipulate the string's characters.

A string may be declared using a pointer just like it was with a char array, but now we use a pointer variable (no square brackets) instead of an array variable. The string may be initialized when it is declared, or it may be assigned later. The string itself will be stored in memory, and the pointer will be given the address of the first character of the string.



Example



Command line arguments:

Command line arguments is an important concept in c programming.

At the time of running a program we can pass the arguments to the main function. These arguments are called command line arguments.

Sometimes we need to pass arguments from the command line to the program a set of inputs.

Command line arguments are used to supply parameters to the program when it is invoked.

It is mostly used when you need to control your program from the console.

These arguments are passed to the main() method.

argc and argv

To facilitate the main() function to accept arguments from the command line, you should define two arguments in the main() function – argc and argv[].
argc (argument count)refers to the number of arguments passed and argv[](argument vector) is a pointer array that points to each argument passed to the program.
Syntax:

```
int main(int argc, char *argv[]) {
        ...
        ...

        return 0;
}
```

The argc argument should always be non-negative. The argv argument is an array of character pointers to all the arguments, argv[0] being the name of the program. After that till "argv [argc - 1]", every element is a command-line argument.

Open any text editor and save the following code as "hello.c" −

```
#include <stdio.h>

int main (int argc, char * argv[]){
   printf("Hello %s", argv[1]);
   return 0;
}
```

The program is expected to fetch the name from argv[1] and use it in the printf() statement.

Instead of running the program from the Run menu of any IDE, compile it from the command line.

Implementation:

```
C:\Users\user>gcc -c hello.c -o hello.o
```

Build the executable –

```
C:\Users\user>gcc -o hello.exe hello.o
```

Pass the name as a command line argument –

```
C:\Users\user>hello Prakash
Hello Prakash
```

Pointer:

A pointer is a variable that stores the memory address of another variable. Instead of holding a data value directly, it holds the address where the value is stored in memory. Pointer are powerful features in C that allows you to directly manage memory and work with variable addresses. There are 2 important operators that we will use in pointers concepts i.e.

• Dereferencing operator (*) used to declare pointer variable and access the value stored in the address.

• Address operator (&) used to returns the address of a variable or to access the address of a variable to a pointer.

A. Pointer Declaration

To declare a pointer, we use the (*) dereference operator before its name. In pointer declaration, we only declare the pointer but do not initialize it.

Declare a pointer by specifying the data type it points to, followed by an asterisk (*) and pointer name for Example. Data Type *Pointer name (int *ptr)

B. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We use the (&) address of operator to get the memory address of a variable and then store it in the pointer variable.

To initialize a pointer, you assign it the address of a variable using the address of operator (&) for Example. int var = 10; ptr = &var

C. Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same (*) dereferencing operator that we used in the pointer declaration.

To access the value stored at the memory address a pointer holds, you use the dereference operator (*) for

Example int value = *ptr;

D. Pointer Reference

In C, referencing a pointer means obtaining the address of a variable and storing it in a pointer variable. This is done using the address operator (&).

How to reference a pointer in C

Use the address operator (&) to get the address of a variable

Store the address in a pointer variable

Example

int x = 5;

int *pointer = &x;

In this example, &x references the memory address of x and stores it in the pointer variable.

Advantages of Pointers

Following are the major advantages of pointers in C:

- Pointers are used for dynamic memory allocation and de-allocation.
- An Array or a structure can be accessed efficiently with pointers
- Pointers are useful for accessing memory locations.
- Pointers are used to form complex data structures such as linked lists, graphs, trees, etc.
- Pointers reduce the length of the program and its execution time as well.

Use of Pointer

1. Memory Management
   pointer give you control over memory allocation and de-allocation
2. Passing by Reference
   You can pass pointer to functions to allow the function to modify the original data.
3. Dynamic data Structure
   Pointer are essential for creating complex data structures like linked lists, trees, and graph

Data model-

Value model Vs reference model

a "value model" refers to a variable storing the actual data value directly, while a "reference model" stores the memory address of where the data is located.

Types of pointer

Depending on the parameters, pointers can be divided into a wide variety of types. The following types of pointers are based on the type of variable that is kept in the memory location that the pointer is pointing to.

Integer Pointers

The memory location of an integer variable is stored in integer pointers.

Syntax

```
int* intPointer;
```

Array Pointer

The initial element of an array may be referenced by array pointers.

Syntax

```
int arr[5];
int* arrPointer = arr;
```

Structure Pointer

To point to a user-defined datatype, structure use a structure pointer.

Syntax

```
struct MyStruct {
 int data;
};
struct MyStruct* structPointer;
```

Function Pointers

You can call functions indirectly by using function pointers, that save the address of a function.

Syntax

```
int (*funcPointer)(int, int);
```

Double Pointers

Double pointers are pointers to pointers and are frequently employed in linked data structures or multi-dimensional arrays.

Syntax

```
int doublePointer;
```

NULL Pointer

A NULL pointer is used to signify that a pointer has not been initialized; it points to nowhere in memory.

Syntax

```
int* nullPointer = NULL;
```

void Pointer

In general, void pointers can point to any type of data.

Syntax

void* voidPointer;

wild Pointers

wild pointers are dangling or uninitialized pointers that point to arbitrary locations in memory.

Syntax

int *ptr;

char *str;

constant Pointers

Constant pointers direct memory access to a fixed place.

Syntax

const int* constPointer;

Pointer to Constant

A pointer to a constant point to data that the pointer cannot change.

Syntax

int const * pointerToConst;

far Pointer

Early versions of the x86 architecture used far pointers to access extended memory. Today's programming hardly makes use of them.

Syntax

char far *s;

Dangling Pointer

A dangling pointer is a pointer that frequently occurs after the memory it points to has been released or deallocated and points to an incorrect address in memory.

Syntax

int *ptr = malloc(sizeof(int)); // ptr points to a valid memory location

free(ptr); // ptr now points to a dangling memory location

{

int a = 10;

int *ptr = &a; // ptr points to a valid memory location

} // a goes out of scope, ptr now points to a dangling memory location

huge Pointer

In the early x86 architecture, huge pointers were utilized to access enormous memory models. Today's programming hardly makes use of them.

Syntax

int huge *p;

complex Pointer

Depending on the use case, complex pointers may refer to pointers having complex data structures that need a particular syntax.

Syntax

int **p;

near Pointer

Early versions of the x86 architecture employed near pointers to access tiny memory models. Today's programming hardly makes use of them.

Syntax

char near *string;

normalized Pointer

In low-level systems programming, normalized pointers are frequently used to guarantee that the address of the pointer is within the legal address space.

Syntax

int *ptr = _mkptr(0x12, 0x34);

File Pointer

In file I/O operations, file pointers are used to keep track of the current location in a file.

Syntax

FILE* filePointer;

Pointer Arithmatic-

We can perform arithmetic operations on the pointers like addition, subtraction, etc.
However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer. In pointer-from-pointer subtraction, the result will be an integer value.
Following arithmetic operations are possible on the pointer in C language:

o Increment

o Decrement

o Addition

o Subtraction

o Comparison

Incrementing a Pointer

If we incerement a pointer by 1, the pointer will start pointing to the immediate next location.
This is different from the general arithmetic because the value of the pointer will get increased by the size of the data type to which the pointer is pointing.
The rule to increment the pointer is given below.
new_address=current_sddress+i*size_of(data type)

where I is the number by which the pointer get increased.

We can traverse an array by using pointer.

Example:

Int a=10;

Int *p;

P=&a;

P=p+1;

Here suppose address of a is 2010 which is stored in pointer variable p. we perform increment operation on pointer variable p. then new value of p becomes p=p+1*22010+2=2012

Decrementing pointer

Like increment, we can decrement a pointer variable. If we decrement a pointer, it will start pointing to the previous location. The formulae of decrementing the pointer is given below.

new_address=current addres-i*size_of(data type)

example

int a=10;

int*p;

p=&a;

p=p-1;

here suppose address of **a** is 2010 which is stored in pointer variable p.

we perform decrement operation on pointer variable p. then new value of p becomes.

P=p-1*2=2010-2=2008

Pointer addition

We can add **a** value to the pointer variable. The formulae of adding value to pointer is given below.

new_address=current_address+(number*size_of(data type)

example

float a=10;

float*p;

p=&a;

p=p+3;

here suppose address of **a** is 2410 which is stored in pointer variable p. we add value 3 in pointer variable p. then new value of p between the address.

P=p+(3*4)=2410+12=2422

We can not add two pointers in each other.

Pointer substraction

Like pointer addition, we can substract a value from the pointer variable.

Subtracting any number from a pointer will give an address.

The formula of subtracting value from the pointer variable is given below.

New_address=current_address-(number*size_of(data type))

Example

Float a=10;

Float*p;

P=&a;

P=p-2;

Here suppose address of **a** is 2410 which stored in pointer variable p. we can subtract two pointers from each other to get offset.