Module - V

Pointers: Declarations and initialization of pointer variables ,Accessing pointer variables, Passing to a function. Operations on pointers, pointer and arrays. Array of pointers, Pointer to Functions. Data Files: Open, close, create, process unformatted data files.

Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

type *var-name;

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations −

int *ip;	pointer to an integer
double *dp;	pointer to a double
float *fp;	pointer to a float
char *ch	pointer to a character

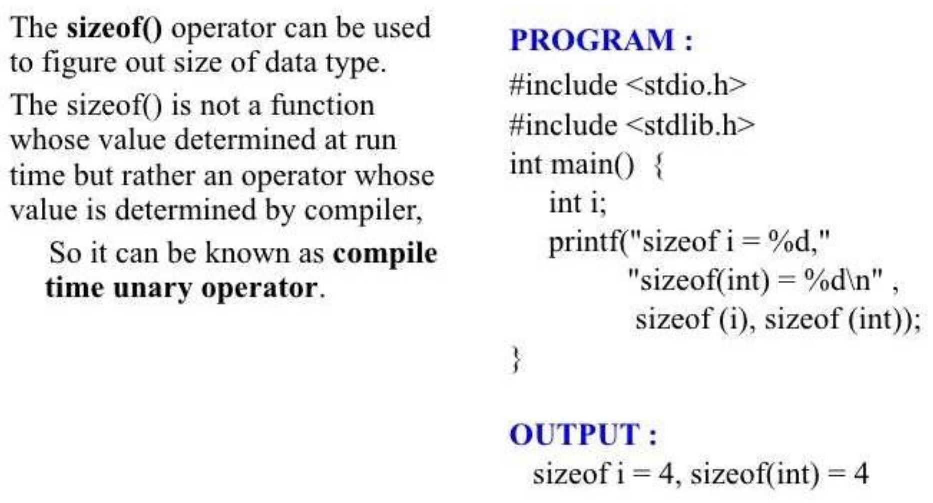
The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Pointer to function

C programming allows passing a pointer to a function. To do so, simply declare the function parameter as a pointer type.

Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function −

Sizeof()



The **sizeof()** operator can be used to figure out size of data type.

The sizeof() is not a function whose value determined at run time but rather an operator whose value is determined by compiler,

So it can be known as **compile time unary operator**.

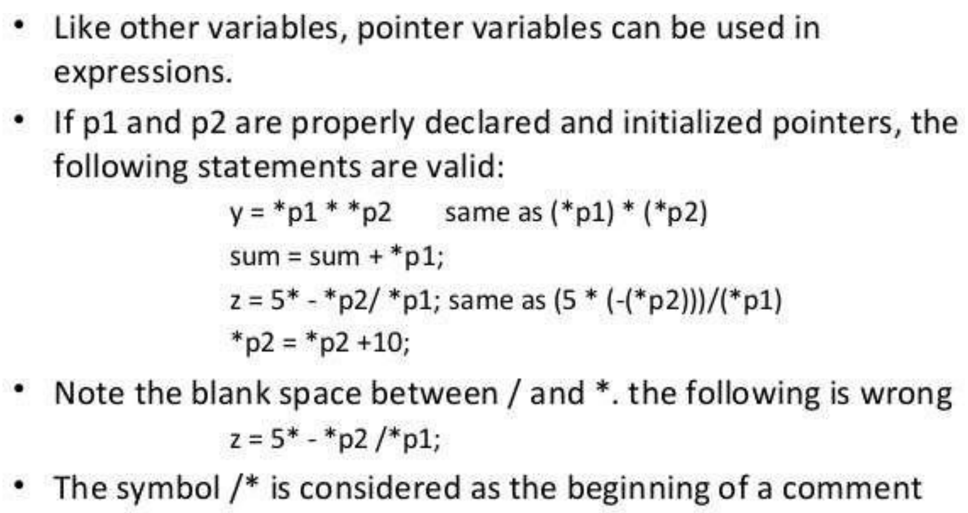
PROGRAM :

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

OUTPUT :

```
  sizeof i = 4, sizeof(int) = 4
```

Pointer Expressions and Pointer Arithmetic



- Like other variables, pointer variables can be used in expressions.
- If p1 and p2 are properly declared and initialized pointers, the following statements are valid:

```
y = *p1 * *p2       same as (*p1) * (*p2)
sum = sum + *p1;
z = 5* - *p2/ *p1; same as (5 * (-(*p2)))/(*p1)
*p2 = *p2 +10;
```

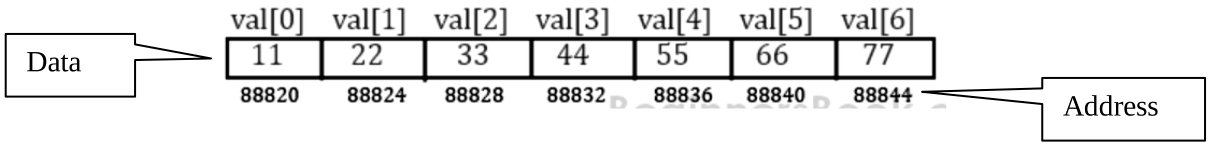
- Note the blank space between / and *. the following is wrong

```
z = 5* - *p2 /*p1;
```

- The symbol /* is considered as the beginning of a comment

Pointer and Array

All the elements in array are stored in consecutive contiguous memory locations in the memory



We can access entire array through single pointer variable using its address location. Since the array elements are stored in to consecutive location.

```
#include <stdio.h>
void main( )
{
    int *p;
    int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;
    p = &val[0];
   for ( int i = 0 ; i<7 ; i++ )
   {
      printf("val[%d]: value is %d and address is %p\n", i, *p, p);
      p++;
   }
}
```

Output:

```
val[0]: value is 11 and address is 88820
val[1]: value is 22 and address is 88824
val[2]: value is 33 and address is 88828
val[3]: value is 44 and address is 88832
val[4]: value is 55 and address is 88836
val[5]: value is 66 and address is 88840
val[6]: value is 77 and address is 88844
```

While using pointers with array, the data type of the pointer must match with the data type of the array.

Array of pointers:

“Array of pointers” is an array of the pointer variables. It is also known as pointer arrays.

Syntax:

```
int *var_name[array_size];
```

Declaration of an array of pointers:

```
int *ptr[3];
```

We can make separate pointer variables which can point to the different values or we can make one integer array of pointers that can point to all the values.

Example: Array of pointers.

```
#include <stdio.h>
const int SIZE = 3;
void main()
{
                                // creating an array
   int arr[] = { 1, 2, 3 };

                                  // we can make an integer pointer array to
                                  // storing the address of array elements
   int i, *ptr[SIZE];
   for (i = 0; i < SIZE; i++) {
                                         // assigning the address of integer.
      ptr[i] = &arr[i];
   }
                                        // printing values using pointer
   for (i = 0; i < SIZE; i++) {

      printf("Value of arr[%d] = %d\n", i, *ptr[i]);
   }
}
```

Output:

```
Value of arr[0] = 1
Value of arr[1] = 2
Value of arr[2] = 3
```

Data Files

The last chapter explained the standard input and output devices handled by C programming language. This chapter cover how C programmers can create, open, close text or binary files for their data storage.

A file represents a sequence of bytes, regardless of it being a text file or a binary file. C programming language provides access on high level functions as well as low level OSlevelOSlevel calls to handle file on your storage devices. This chapter will take you through the important calls for file management.

Opening Files

You can use the **fopen** function to create a new file or to open an existing file. This call will initialize an object of the type **FILE**, which contains all the information necessary to control the stream. The prototype of this function call is as follows −

```
FILE *fopen( const char * filename, const char * mode );
```

Here, **filename** is a string literal, which you will use to name your file, and access **mode** can have one of the following values −

Sr.No.	Mode & Description
1	**r -** Opens an existing text file for reading purpose.
2	**w-** Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
3	**a -** Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
4	**r+** Opens a text file for both reading and writing.
5	**w+** Opens a text file for both reading and writing. It first truncates the file to zero length if it exists, otherwise creates a file if it does not exist.
6	**a+** Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Closing a File

To close a file, use the fclose function. The prototype of this function is −

```
int fclose( FILE *fp );
```

The **fclose−−** function returns zero on success, or EOF if there is an error in closing the file. This function actually flushes any data still pending in the buffer to the file, closes the file, and releases any memory used for the file. The EOF is a constant defined in the header file stdio.h.

There are various functions provided by C standard library to read and write a file, character by character, or in the form of a fixed length string.

Writing a File

Following is the simplest function to write individual characters to a stream −

int fputc(int c, FILE *fp);

The function fputc writes the character value of the argument c to the output stream referenced by fp. It returns the written character written on success otherwise EOF if there is an error. You can use the following functions to write a null-terminated string to a stream −

int fputs(const char *s, FILE *fp);

The function fputs writes the string s to the output stream referenced by fp. It returns a non- negative value on success, otherwise EOF is returned in case of any error. You can use int fprintfFILE* fp,constchar* format,...FILE* fp,constchar* format,... function as well to write a string into a file. Try the following example.

Make sure you have /tmp directory available. If it is not, then before proceeding, you must create this directory on your machine.

```
#include
<stdio.h> main()
{
  FILE *fp;
  fp = fopen("/tmp/test.txt", "w+");
  fprintf(fp, "This is testing for
  fprintf...\n"); fputs("This is testing for
  fputs...\n", fp); fclose(fp);
}
```

When the above code is compiled and executed, it creates a new file test.txt in /tmp directory and writes two lines using two different functions. Let us read this file in the next section.

Reading a File

Given below is the simplest function to read a single character from a file −

int fgetc(FILE * fp);

The fgetc function reads a character from the input file referenced by fp. The return value is the character read, or in case of any error, it returns EOF. The following function allows to read a string from a stream −

char *fgets(char *buf, int n, FILE *fp);

The functions fgets reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer buf, appending a null character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including the new line character. You can also use int fscanfFILE* fp,constchar* format,...FILE* fp,constchar* format,... function to read strings from a file, but it stops reading after encountering the first space character.

```
#include
<stdio.h> main()
{
  FILE *fp;
  char buff[255];
  fp = fopen("/tmp/test.txt", "r");
  fscanf(fp, "%s", buff);
  printf("1 : %s\n", buff );
  fgets(buff, 255,
  (FILE*)fp); printf("2:
  %s\n", buff ); fgets(buff,
  255, (FILE*)fp);
  printf("3: %s\n", buff );
  fclose(fp);
}
```

When the above code is compiled and executed, it reads the file created in the previous section and produces the following result −

```
1 : This
2: is testing for fprintf...
3: This is testing for fputs...
```

Let's see a little more in detail about what happened here. First, fscanf read just This because after that, it encountered a space, second call is for fgets which reads the remaining line till it encountered end of line. Finally, the last call fgets reads the second line completely.

Binary I/O Functions

There are two functions, that can be used for binary input and output −

size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t number_of_elements, FILE *a_file);

Both of these functions should be used to read or write blocks of memories - usually arrays or structures.

Random Access to File

There is no need to read each record sequentially, if we want to access a particular record.C supports these functions for random access file processing.

1. fseek()
2. ftell()
3. rewind()

fseek():
This function is used for seeking the pointer position in the file at the specified byte.
Syntax: fseek(file pointer, displacement, pointer position);
Where
file pointer ---- It is the pointer which points to the file.
displacement ---- It is positive or negative.This is the number of bytes which are skipped backward (if negative) or forward(if positive) from the current position.This is attached with L because this is a long integer.

ftell()

This function returns the value of the current pointer position in the file.The value is count from the beginning of the file.