

UNIT 5 PPS

Pointers: A pointer is a variable that stores the memory address of another variable as its value. A pointer variable points to a data type (like int) of the same type, and is created with the * operator.

Syntax:

```
datatype *var_name;
```

Uses/Applications of pointers:

- **To pass arguments by reference.**
- For accessing array elements.
- **Dynamic memory allocation.**
- To implement data structures.(eg Linked list)
- To do system-level programming where memory addresses are useful.

Pointer Example

```
#include<stdio.h>
void main()
{
    int a=50;
    int *p;
    p=&a;//stores the address of number variable
    printf("Address of p variable is %d\n",p);
    printf("Value of p variable is %d \n",*p);
    printf("Address of a variable is %d \n",&a);
}
```

Output:

Address of p variable is 2018028084

Value of p variable is 50

Address of a variable is 2018028084

Static Memory Allocation: In Static Memory Allocation size is fixed which can waste memory. Example: `int a[100];` reserves space for 100 integers if only 50 integers are used rest memory is wasted.

So we use Dynamic Memory Allocation.

Dynamic Memory Allocation(DMA) can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under `<stdlib.h>` header file to facilitate dynamic memory allocation in C programming. They are:

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

Both `malloc` and `calloc` are used for dynamic memory allocation in C with some differences.

Difference Between `malloc()` and `calloc()` in C

Basis of differences	Malloc	Calloc
Define	Memory Allocation is the full form of <code>malloc</code> which means a single dynamic memory block is allocated during runtime.	Contiguous Allocation is the full form of <code>calloc</code> which means multiple memory blocks are allocated to a single variable.
Parameters	Malloc takes only one parameter.	Calloc takes two parameters.
Initialization	When we try to access the block of memory allocated, the garbage value is returned because the <code>malloc()</code> function doesn't initialize allocated memory.	When we try to access the memory blocks we get the result as '0' because <code>calloc</code> initializes the allocated memory block to '0'.
Speed	Malloc does its job quickly.	Calloc is slow in comparison to Malloc.
Syntax	<pre>ptr_variable = (cast-type*) malloc(byte-size)</pre> <p>For Example: <i><code>ptr = (int*) malloc(100 * sizeof(int));</code></i></p>	<pre>ptr_variable = (cast-type*) calloc(n,byte-size)</pre> <p>For Example: <i><code>ptr = (int*) calloc(n, sizeof(int));</code></i></p>

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

Syntax of realloc() in C

```
ptr = realloc(ptr, newSize);
```

where `ptr` is reallocated with new size `'newSize'`.

Free():

Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by `ptr`.

<p>Program to show use of malloc, realloc and free</p> <pre> #include<stdio.h> #include<stdlib.h> void main() { int *p,*q,n,i,sum=0; printf("enter num of integers\n"); scanf("%d",&n); p=(int*)malloc(n*sizeof(int)); printf("enter number \n"); for(i=0;i<n;i++) { scanf("%d",&p[i]); } printf("Entered numbers \n"); for(i=0;i<n;i++) { printf("%d\t",p[i]); } q=realloc(p,100*sizeof(int)); //reallocate memory free(p); } Output: Enter num of integers 5 Enter number 12 34 55 67 78 Entered numbers 12 34 55 67 78 </pre>	<p>Program to show use of calloc</p> <pre> #include<stdio.h> #include<stdlib.h> void main() { int *p,n,i,sum=0; printf("enter num of integers\n"); scanf("%d",&n); p=(int*)calloc(n,sizeof(int)); printf("enter numbers \n"); for(i=0;i<n;i++) { scanf("%d",&p[i]); } printf("Entered numbers \n"); for(i=0;i<n;i++) { printf("%d\t",p[i]); } free(p); } Output: enter num of integers 5 enter numbers 2 3 4 5 67 Entered numbers 2 3 4 5 67 </pre>	<p>to show</p>
---	---	----------------

Self Referential Structures using pointers

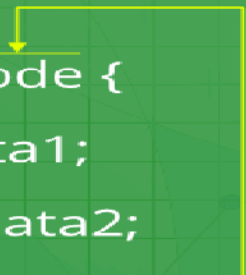
The self-referential structure is a structure that points to the same type of structure. It contains one or more pointers that ultimately point to the same structure.

- Structures are a **user-defined** data structure type in C and C++.
- The main benefit of creating structure is that it can hold the different predefined data types.

Why do we require a Self-referential structure?

Self-referential structure plays a vital role in the linked list, trees, graphs, and many more data structures. By using the structure, we can easily implement these data structures efficiently.

Self Referential Structures



```
struct node {  
    int data1;  
    char data2;  
    struct node* link;  
};
```

Linked List

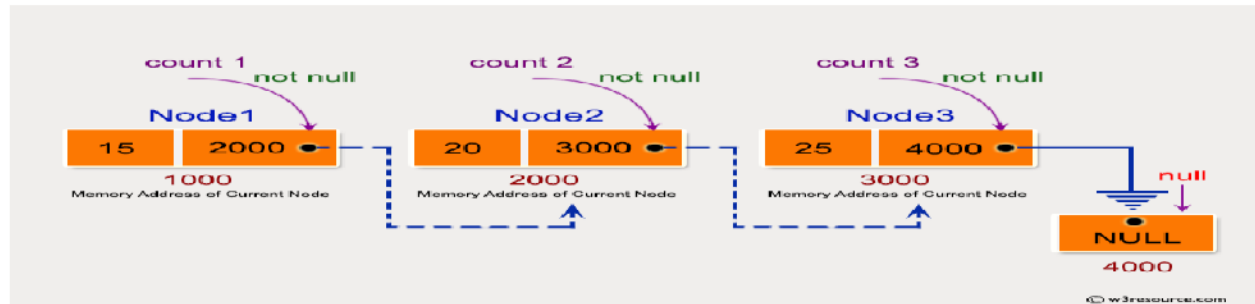
What is Linked List in C?

A Linked List is a **linear data structure**. Every linked list has two parts, the data section and the address section that holds the address of the next element in the list, which is called a node

Each node in a list consists of at least two parts:

- A Data Item (we can store integers, strings, or any type of data).
- Pointer (Or Reference) to the next node (connects one node to another) or An address of another node

```
// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```



Advantages Of Linked List: A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give the initial size of the linked list.

Macros in c:

A macro is a piece of code in a program that is replaced by the value of the macro. Macro is defined by #define directive. Whenever a macro name is encountered by the compiler, it replaces the name with the definition of the macro.

Example1

```
include<stdio.h>

// This is macro definition
#define PI 3.14

void main()
{
    // declaration and initialization of radius
    int radius = 5;
    // declaration and calculating the area
    float area = PI * (radius*radius);

    // Printing the area of circle
    printf("Area of circle is %f", area);
}
```

```
Area of circle is 78.500000
```

C program to illustrate function macros

```
#include <stdio.h>

// Macro definition
#define AREA(l, b) (l * b)

void main()
{
    // Given lengths l1 and l2
    int l1 = 10, l2 = 5, area;

    // Find the area using macros
    area = AREA(l1, l2);

    printf("Area of rectangle %d", area);
}
```

Output:Area of rectangle 50

File input/Output(file handling) in C:

File handling allows us to preserve/store the information/data generated after we run the program. This saves time of the user as he can store important information.

There are two types of files :

1.Binary Files (store data in binary form 0 and 1)

- 2.Text Files (store data in alphabets and numbers which are easily understood by human beings.)

File opening modes in c

1. r - open a file in read mode.
2. w - opens or create a text file in write mode.
3. a - opens a file in append mode.
4. r+ - opens a file in both read and write mode.
5. a+ - opens a file in both read and write mode.
- 6.w+ - opens a file in both read and write mode

Mode	Read	Write	Create New File*	Truncate
r	Yes	No	No	No
w	No	Yes	Yes	Yes
a	No	Yes	Yes	No
r+	Yes	Yes	No	No
w+	Yes	Yes	Yes	Yes
a+	Yes	Yes	Yes	No
*Creates a new file if it doesn't exist.				

Write data to a text file

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main()
{
    int num;
    char name[20];
    FILE *fptr;
    fptr = (fopen("c://program.txt","a "));
    if(fptr == NULL)

    {
        printf("Error!");
        exit(1);
    }
    printf("Enter rollno ");
    scanf("%d",&num);
    printf("Enter name: ");
    scanf("%s",name);

    fprintf(fptr,"%d",num);
    fprintf(fptr,"%s",name);
    fclose(fptr);
    getch();
}
```

Read data from a text file

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h> // For exit()
void main()
{
    FILE *fptr;
    char filename[100], c;
    // Open file
    fptr = fopen("c:\\program.txt, "r");
    if (fptr == NULL)
    {
        printf("Cannot open file \n");
        exit(0);
    }
    // Read contents from file
    c = fgetc(fptr);
    while (c != EOF)
    {
        printf ("%c", c);
        c = fgetc(fptr);
    }
    fclose(fptr);

    getch();
}
```

Copy one file to another

Program to copy one file contents to another in c

```
#include<stdio.h>
void main()
{
    int ch;
    FILE *fp,*fq;
    fp=fopen("source.txt","r");
    fq=fopen("backup.txt","w");
    if(fp==NULL||fq==NULL)
        printf("File does not exist..");
    else
        while((ch=fgetc(fp))!=EOF)
        {
            fputc(ch,fq);
        }
    printf("File copied.....");
}
```

File Operations:

- 1.Read data from file:This operation read data from a file.
- 2.Write data to file:This operation writes the data to a file.
- 3.Copy data from one file to another: This operation copies a file to another file.

More Functions for C File Operations

The following table lists some more functions that can be used to perform file operations or assist in performing them.

Functions	Description
<code>fopen()</code>	It is used to create a file or to open a file.
<code>fclose()</code>	It is used to close a file.
<code>fgets()</code>	It is used to read a file.
<code>fprintf()</code>	It is used to write blocks of data into a file.
<code>fscanf()</code>	It is used to read blocks of data from a file.
<code>getc()</code>	It is used to read a single character to a file.
<code>putc()</code>	It is used to write a single character to a file.
<code>fseek()</code>	It is used to set the position of a file pointer to a mentioned location.
<code>ftell()</code>	It is used to return the current position of a file pointer.
<code>rewind()</code>	It is used to set the file pointer to the beginning of a file.

Command line arguments

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using `main()` function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

```
1. #include <stdio.h>
2. void main(int argc, char *argv[] ) {
3.
4.     printf("Program name is: %s\n", argv[0]);
5.
6.     if(argc < 2){
7.         printf("No argument passed through command line.\n");
8.     }
9.     else{
10.        printf("First argument is: %s\n", argv[1]);
11.    }
12.}
```

Run this program as follows in Windows from command line:

program.exe hello

Output:

```
Program name is: program
First argument is: hello
```

String and string functions

Strings are an array of characters that terminate with a null character '\0'. The difference between a character array and a string is that, unlike the character array, the string ends with a null character. There are various built-in string functions in the C programming,

It uses header file `string.h`

Example: `char a[]="hello";`

Functions of String:

1)String Length

For example, to get the length of a string, you can use the `strlen()` function:

Example

```
#include <stdio.h>
#include <string.h>
void main()
{
    char a[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    printf("%d", strlen(a));
}
```

Output: **Length is: 26**

2) Concatenate Strings

To concatenate (combine) two strings, you can use the `strcat()` function:

```
#include <stdio.h>
#include <string.h>

void main() {
    char str1[20] = "Hello ";
    char str2[] = "World!";
    strcat(str1, str2);
    printf("%s", str1);
}
```

Output: **Hello World!**

3) Copy Strings

To copy the value of one string to another, you can use the `strcpy()` function:

```
#include <stdio.h>
#include <string.h>

void main() {

    char str1[20] = "Hello World!";
    char str2[20];

    // Copy str1 to str2
    strcpy(str2, str1);
    printf("%s", str2);
}
```

Output:
Hello World!

PREPROCESSOR DIRECTIVES IN C:

As the name suggests, Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C .

There are 4 Main Types of Preprocessor Directives:

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

1. Macros: Macros are pieces of code in a program that is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code. The '#define' directive is used to define a macro.

Simple macros example:

```
#include <stdio.h>
// Macro definition
#define AREA(l, b) (l * b)
void main()
{
    // Given lengths l1 and l2
    int l1 = 10, l2 = 5, area;

    // Find the area using macros
    area = AREA(l1, l2);

    printf("Area of rectangle %d", area);
}
```

Output:Area of rectangle 50

2. File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code program. There are two types of files that can be included by the user in the program:

Header files or Standard files: These files contain definitions of pre-defined functions like `printf()`, `scanf()`, etc. These files must be included to work with these functions. Different functions are declared in different header files.

Example: `#include <stdio.h>`

3. Conditional Compilation

Conditional Compilation directives are a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions. This can be done with the help of the two preprocessing commands '`ifdef`' and '`endif`'.

```
#include <stdio.h>
#define x 10

int main()
{
    #ifdef x
        printf("hello\n");           // this is
compiled as x is defined
    #else
        printf("bye\n");             // this isn't
compiled
    #endif

    return 0;
}
```

4. Other Directives

Apart from the above directives, there are two more directives that are not commonly used. These are:

#undef Directive: The `#undef` directive is used to undefine an existing macro. This directive works as:

`#undef LIMIT`