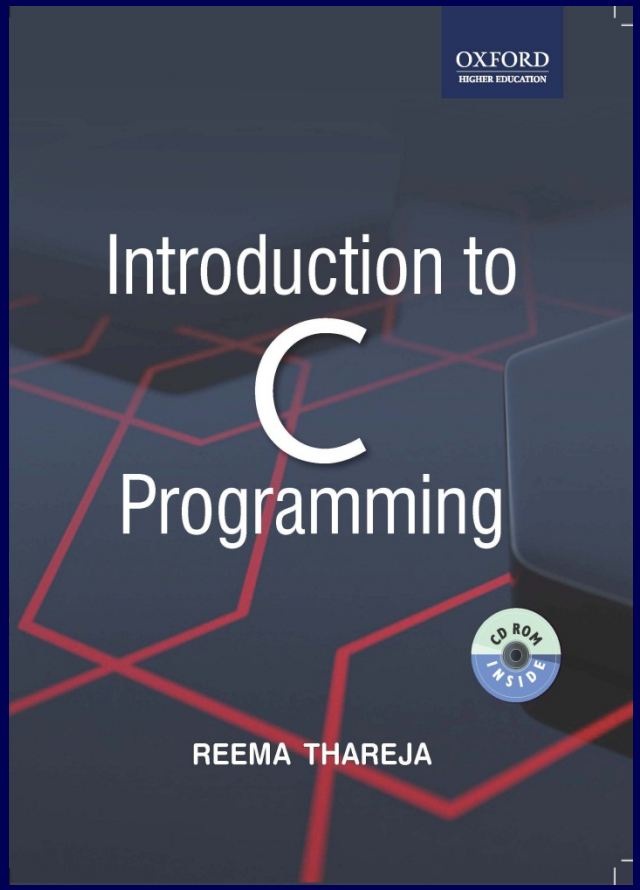
Introduction to C Programming

Reema Thareja, Assistant Professor,
Institute of Information Technology and Management



CHAPTER 7

POINTERS

UNDERSTANDING THE COMPUTER'S MEMORY

- Every computer has a primary memory. All our data and programs need to be placed in the primary memory for execution.
- The primary memory or RAM (Random Access Memory which is a part of the primary memory) is a collection of memory locations (often known as cells) and each location has a specific address. Each memory location is capable of storing 1 byte of data
- Generally, the computer has three areas of memory each of which is used for a specific task. These areas of memory include- stack, heap and global memory.
- **Stack-** A fixed size of stack is allocated by the system and is filled as needed from the bottom to the top, one element at a time. These elements can be removed from the top to the bottom by removing one element at a time. That is, the last element added to the stack is removed first.
- **Heap-** Heap is a contiguous block of memory that is available for use by the program when need arise. A fixed size heap is allocated by the system and is used by the system in a random fashion.
- When the program requests a block of memory, the dynamic allocation technique carves out a block from the heap and assigns it to the program.
- When the program has finished using that block, it returns that memory block to the heap and the location of the memory locations in that block is added to the free list.

UNDERSTANDING THE COMPUTER'S MEMORY contd.

- **Global Memory-** The block of code that is the main() program (along with other functions in the program) is stored in the global memory. The memory in the global area is allocated randomly to store the code of different functions in the program in such a way that one function is not contiguous to another function. Besides, the function code, all global variables declared in the program are stored in the global memory area.
- **Other Memory Layouts-** C provides some more memory areas like- text segment, BSS and shared library segment.
- The text segment is used to store the machine instructions corresponding to the compiled program. This is generally a read-only memory segment
- BSS is used to store un-initialized global variables
- Shared libraries segment contains the executable image of shared libraries that are being used by the program.

INTRODUCTION

- Every variable in C has a name and a value associated with it. When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable. The size of the allocated block depends on the type of the data.

  int x = 10;

- When this statement executes, the compiler sets aside 2 bytes of memory to hold the value 10. It also sets up a symbol table in which it adds the symbol x and the relative address in memory where those 2 bytes were set aside.
- Thus, every variable in C has a value and an also a memory location (commonly known as address) associated with it. Some texts use the term *rvalue* and *lvalue* for the value and the address of the variable respectively.
- The rvalue appears on the right side of the assignment statement and cannot be used on the left side of the assignment statement. Therefore, writing 10 = k; is illegal.

DECLARING POINTER VARIABLES

- Actually pointers are nothing but memory addresses.
- A pointer is a variable that contains the memory location of another variable.
- The general syntax of declaring pointer variable is

  data_type *ptr_name;

  Here, data_type is the data type of the value that the pointer will point to. For example:

  int *pnum; char *pch; float *pfnum;

  int x= 10;

  int *ptr = &x;

  The '*' informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable.

  The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr.

DE-REFERENCING A POINTER VARIABLE

- We can "dereference" a pointer, i.e. refer to the value of the variable to which it points by using unary '*' operator as in *ptr. That is, *ptr = 10, since 10 is value of x.

```
#include<stdio.h>
int main()
{
        int num, *pnum;
        pnum = &num;
        printf("\n Enter the number : ");
        scanf("%d", &num);
        printf("\n The number that was entered is : %d", *pnum);
        return 0;
}
```

OUTPUT:

```
Enter the number : 10
The number that was entered is : 10
```

POINTER EXPRESSIONS AND POINTER ARITHMETIC

- Pointer variables can also be used in expressions. For ex,

```
int num1=2, num2= 3, sum=0, mul=0, div=1;
int *ptr1, *ptr2;
ptr1 = &num1, ptr2 = &num2;
sum = *ptr1 + *ptr2;
mul = sum * *ptr1;
*ptr2 +=1;
div = 9 + *ptr1/*ptr2 - 30;
```

- We can add integers to or subtract integers from pointers as well as to subtract one pointer from the other.
- We can compare pointers by using relational operators in the expressions. For example p1 > p2 , p1==p2 and p1!=p2 are all valid in C.

When using pointers, unary increment (++) and decrement (--) operators have greater precedence than the dereference operator (*). Therefore, the expression
*ptr++ is equivalent to *(ptr++). So the expression will increase the value of ptr so that it now points to the next element.
In order to increment the value of the variable whose address is stored in ptr, write (*ptr)++

NULL POINTERS

A *null pointer* which is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address.

To declare a null pointer you may use the predefined constant NULL,

```
int *ptr = NULL;
```

You can always check whether a given pointer variable stores address of some variable or contains a null by writing,

```
if ( ptr == NULL)
{       Statement block;
}
```

Null pointers are used in situations if one of the pointers in the program points somewhere some of the time but not all of the time. In such situations it is always better to set it to a null pointer when it doesn't point anywhere valid, and to test to see if it's a null pointer before using it.

GENERIC POINTERS

- A generic pointer is pointer variable that has void as its data type.
- The generic pointer, can be pointed at variables of any data type.
- It is declared by writing

  void *ptr;

- You need to cast a void pointer to another kind of pointer before using it.
- Generic pointers are used when a pointer has to point to data of different types at different times. For ex,

```
#include<stdio.h>
int main()
{ int x=10;
        char ch = 'A';
        void *gp;
        gp = &x;    printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
        gp = &ch;   printf("\n Generic pointer now points to the character %c", *(char*)gp);
    return 0;
} OUTPUT:
```

- Generic pointer points to the integer value = 10
- Generic pointer now points to the character = A

PASSING ARGUMENTS TO FUNCTION USING POINTERS

The calling function sends the addresses of the variables and the called function must declare those incoming arguments as pointers. In order to modify the variables sent by the caller, the called function must dereference the pointers that were passed to it. Thus, passing pointers to a function avoid the overhead of copying data from one function to another.

```
#include<stdio.h>
int main()
{
        int num1, num2, total;
        printf("\n Enter two numbers : ");
        scanf("%d %d", &num1, &num2);
        sum(&num1, &num2, &total);
        printf("\n Total = %d", total);
        getch();
        return 0;
}
void sum ( int *a, int *b, int *t)
{
        *t = *a + *b;
 }
```

POINTERS AND ARRAYS

```
int arr[5] = {1, 2, 3, 4, 5};
int *ptr = &arr[0];
ptr++;
printf("\n The value of the second element of the array is %d", *ptr);
```

Write a program to read and display an array of n integers

```
#include<stdio.h>
int main()
{
        int i, n;
        int arr[10], *parr;
        parr = arr;
        printf("\n Enter the number of elements : ");
        scanf("%d", &n);
        for(i=0; i <n; i++)
                        scanf("%d", (parr+i));
        for(i=0; i <n; i++)
                printf("\n arr[%d] = %d", i, *(parr+i));
}
```

POINTERS AND TWO DIMENSIONAL ARRAY

- Individual elements of the array mat can be accessed using either:
  mat[i][j] or *(*(mat + i) + j) or*(mat[i]+j);
- See pointer to a one dimensional array can be declared as,
  int arr[]={1,2,3,4,5};
  int *parr;
  parr=arr;
- Similarly, pointer to a two dimensional array can be declared as,
  int arr[2][2]={{1,2},{3,4}};
  int (*parr)[2];
  parr=arr;
- Look at the code given below which illustrates the use of a pointer to a two dimensional array.
-

```
#include<stdio.h>
main()
{       int arr[2][2]={{1,2}.{3,4}};
        int i, (*parr)[2];
        parr=arr;
        for(i=0;i<2;i++)
        {       for(j=0;j<2;j++)
                        printf(" %d", (*(parr+i))[j]);
        }
}
```

OUTPUT

1 2 3 4

POINTERS AND STRINGS

- Now, consider the following program that prints a text.

```
#include<stdio.h>
main()
{ char str[] = “Oxford”;
                char *pstr = str;
  printf(“\n The string is : “);
  while( *pstr != ‘\0’)
  {     printf(“%c’, *pstr);
        pstr++;
  }
}
```

- In this program we declare a character pointer *pstr to show the string on the screen. We then "point" the pointer pstr at str. Then we print each character of the string in the while loop. Instead of using the while loop, we could have straight away used the function puts(), like

```
puts(pstr);
```

- Consider here that the function prototype for puts() is:
- int puts(const char *s); Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. Note that the address of the string is passed to the function as an argument.

POINTERS AND STRINGS CONTD.

- Consider another program which reads a string and then scans each character to count the number of upper and lower case characters entered

```
#include<stdio.h>
int main()
{ char str[100], *pstr;
  int upper = 0, lower = 0;
  printf("\n Enter the string : ");
  gets(str);
  pstr = str;
  while( *pstr != '\0')
  {       if(*pstr >= 'A' && *pstr <= 'Z')
                        upper++;
          else if(*pstr >= 'a' && *pstr <= 'z')
                  lower++;
          pstr++;
  }
  printf("\n Total number of upper case characters = %d", upper);
  printf("\n Total number of lower case characters = %d", lower);
}
```

ARRAY OF POINTERS

An array of pointers can be declared as

```
int *ptr[10]
```

The above statement declares an array of 10 pointers where each of the pointer points to an integer variable. For example, look at the code given below.

```
int *ptr[10];
int p=1, q=2, r=3, s=4, t=5;
ptr[0]=&p;
ptr[1]=&q;
ptr[2]=&r;
ptr[3]=&s;
ptr[4]=&t
```

Can you tell what will be the output of the following statement?

```
printf("\n %d", *ptr[3]);
```

Yes, the output will be 4 because ptr[3] stores the address of integer variable s and *ptr[3] will therefore print the value of s that is 4.

POINTER TO FUNCTION

- This is a useful technique for passing a function as an argument to another function.
- In order to declare a pointer to a function we have to declare it like the prototype of the function except that the name of the function is enclosed between parentheses () and an asterisk (*) is inserted before the name.

```
/* pointer to function returning  int */
int (*func)(int a, float b);
```

- If we have declared a pointer to the function, then that pointer can be assigned to the address of the right sort of function just by using its name.
- When a pointer to a function is declared, it can be called using one of two forms:

```
(*func)(1,2);   ORfunc(1,2);
```

```
#include <stdio.h>
void print(int n);
main()
{
        void (*fp)(int);
        fp = print;
        (*fp)(10);
        fp(20);
        return 0;
}
void print(int value)
{       printf("\n %d", value);
}
```

Comparing Function Pointers

```
if(fp >0)                    // check if initialized
{       if(fp == print)
            printf("\n Pointer points to Print");
        else
          printf("\n Pointer not initialized!");
}
```

PASSING A FUNCTION POINTER AS AN ARGUMENT TO A FUNCTION

A function pointer can be passed as a function's calling argument. This is done when you want to pass a pointer to a callback function.

```
include <stdio.h>
int add(int, int);
int subt(int, int);
int operate(int (*operate_fp) (int, int), int, int);
main()
{       int result;
        result = operate(add, 9, 7);
        printf ("\n Addition Result = %d", result);
        result = operate(sub, 9, 7);
        printf ("\n Subtraction Result = %d", result);
}
int add (int a, int b)
{  return (a+b);}
nt subtract (int a, int b){  return (a-b);}
int operate(int (*operate_fp) (int, int), int a, int b)
{       int result;
        result = (*operate_fp)(a,b);
        return result;
```

POINTERS TO POINTERS

- You can use pointers that point to pointers. The pointers in turn, point to data (or even to other pointers). To declare pointers to pointers just add an asterisk (*) for each level of reference.

  For example, if we have:

  int x=10;

  int *px, **ppx;

  px=&x;

  ppx=&px;

  Now if we write,

  printf("\n %d", **ppx);

  Then it would print 10, the value of x.