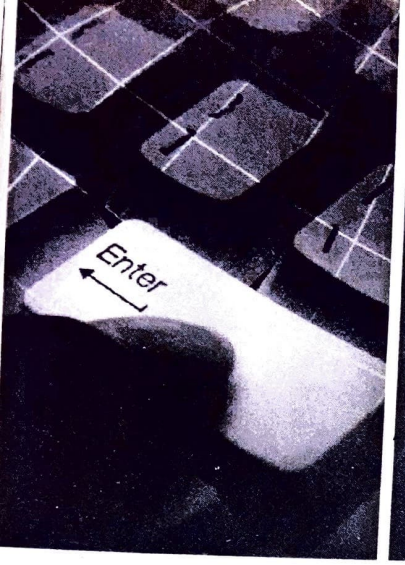


CHAPTER 13

Pointers



LEARNING OBJECTIVE

In this chapter, we will learn about a special variable known as pointers. Pointer variables are used to directly access memory using memory addresses and form a very useful concept in the C language. In this chapter, we will use variables that point to all types of variables —int, float, double, char. Pointers also provide a handy way to access arrays and strings. Moreover, we may also have function pointers in our C programs. Last but not the least, the most powerful use of pointers is to dynamically allocate memory for variables. In this chapter, we will learn about the basics of pointers and then in the subsequent chapters, we will be using them to implement our data structures.

13.1 UNDERSTANDING COMPUTER MEMORY

Every computer has a primary memory. All data and programs need to be placed in the primary memory for execution. The primary memory or RAM 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 (though new computers are able to store 2 bytes of data but in this book we have been talking about locations storing 1 byte of data). Therefore, a char type data needs just 1 memory location, an int type data needs 2 memory locations. Similarly, float and double type data need 4 and 8 memory locations, respectively.

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, i.e., the last element added to the stack is removed first.

When the program has used the variables or data stored in the stack, it can be discarded to enable the stack to be used for other programs to store its data. We have already read a little bit on the system stack in Functions when we discussed Recursion. We will read more about them in the chapter on Stacks.



Stack is the section of memory that is allocated for automatic variables within functions.

Heap It is a contiguous block of memory that is available for use by the program when the need arises. A fixed size heap is allocated by the system and is used by the system in a random fashion.

The addresses of the memory locations in heap that are not currently allocated to the program for use are stored in

a free list. When the program requests a block of memory, the dynamic allocation technique (discussed at the end of this chapter) takes a block from the heap and assigns it to the program. When the program has finished using the block, it returns the memory block to the heap and the location of the memory in that block is added to the free list.

Compared to heaps, a stack is faster but smaller and expensive. When a program begins execution with the main(), all variables declared within main() are allocated space on the stack. Moreover, all the parameters passed to the called function will be stored on the stack.

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 such as 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 (block started by symbol) is used to store un-initialized global variables.
- Shared library segment contains the executable image of shared libraries that are being used by the program.

13.2 INTRODUCTION TO POINTERS

Every variable in C language 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. Let us write a program to find the size of the various data types on your system. (Note the size of integer may vary from one system to another. In 32-bit systems, integer variable is allocated 4 bytes while on 16-bit systems it is allocated 2 bytes).

1. Write a program to find the size of various data types on your system.

```
#include <stdio.h>
int main()
{
   printf("\n The size of short integer is :
   %d", sizeof(short int));
   printf("\n The size of unsigned integer
   is: %d", sizeof(unsigned int));
   printf("\n The size of signed integer is:
   %d", sizeof(signed int));
   printf("\n The size of integer is: %d",
   sizeof(int));
   printf("\n The size of long integer is:
   %d", sizeof(long int));

   printf("\n The size of character is: %d",
   sizeof(char));
   printf("\n The size of unsigned character
   is: %d", sizeof(unsigned char));
   printf("\n The size of signed character
   is: %d", sizeof(signed char));

   printf("\n The size of floating point
   number is: %d", sizeof(float));
   printf("\n The size of double number is:
   %d", sizeof(double));
   return 0;
}
```

Output

```
The size of short integer is: 2
The size of unsigned integer is: 2
The size of signed integer is: 2
The size of integer is: 2
The size of long integer is: 4
The size of character is: 1
The size of unsigned character is: 1
The size of signed character is: 1
The size of floating point number is: 4
The size of double number is: 8
```

Consider the following statement:

```
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 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 (10 in the above statement) and cannot be used on the left side of the assignment statement. Therefore, writing `10 = k;` is illegal. If we write,

```
int x, y;
x = 10;
y = x;
```

Then in this code we have two integer variables `x` and `y`. Compiler reserves memory for integer variable x and stores rvalue 10 in it. When we say `y = x`, then `x` is interpreted as its rvalue (since it is on the right hand side of the

assignment operator '='). Here x refers to the value stored at the memory location set aside for x, in this case 10. After this statement is executed, the rvalue of y is also 10.

You must be wondering why we are discussing addresses and lvalues? Actually pointers are nothing but memory addresses. A *pointer* is a variable that contains the memory location of another variable. Therefore, a pointer is a variable that represents the location of a data item, such as a variable or an array element. Pointers are frequently used in C language as they have a number of useful applications. These applications include the following:

- to pass information back and forth between a function and its reference point
- enable the programmers to return multiple data items from a function via function arguments
- provide an alternate way to access individual elements of the array
- to pass arrays and strings as function arguments
- enable references to functions; so with pointers, the programmer can even pass functions as argument to another function
- to create complex data structures such as trees, linked list, linked stack, linked queue, and graphs
- for dynamic memory allocation of a variable

13.3 DECLARING POINTER VARIABLES

A pointer provides access to a variable by using the address of that variable. A pointer variable is therefore, a variable that stores the address of another variable. The general syntax of declaring pointer variables can be given as follows.

```
data_type *ptr_name;
```

Programming Tip:
The data type of the pointer variable and the variable to which it points must be the same.

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

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

In each of the above statements, a pointer variable is declared to point to a variable of the specified data type. Although all these pointers, `pnum`, `pch`, and `pfnum` point to different data types but they will occupy the same amount of space in memory. However, how much space they occupy will depend on the platform where the code is going to run. To verify this, execute the following code and observe the result.

```
#include <stdio.h>
main()
{
   int *pnum;
   char *pch;
   float *pfnum;
   double *pdnum;
   long *plnum;
   printf("\n Size of integer pointer = %d",
   sizeof(pnum));
   printf("\n Size of character pointer =
   %d", sizeof(pch));
   printf("\n Size of float pointer = %d",
   sizeof(pfnum));
   printf("\n Size of double pointer = %d",
   sizeof(pdnum));
   printf("\n Size of long pointer = %d",
   sizeof(plnum));
}
```

Output

```
Size of integer pointer = 2
Size of character pointer = 2
Size of float pointer = 2
Size of double pointer = 2
Size of long pointer = 2
```

Now let us declare an integer pointer variable and start using it in our program code.

```
int x = 10;
int *ptr;
ptr = &x;
```

In the above statement, `ptr` is the name of pointer variable. The '*' informs the compiler that `ptr` is a pointer variable and the `int` specifies that it will store the address of an integer variable.An integer pointer variable, therefore, *point to* an integer variable. In the last statement, `ptr` is assigned the address of x. The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer `ptr`.

Programming Tip:
A pointer variable can store only the address of a variable.

Consider the memory cells given in Figure 13.1.

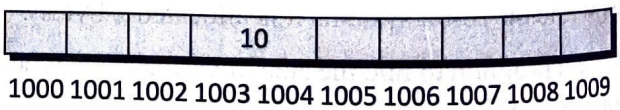


Figure 13.1 Memory representation

Now, since x is an integer variable, it will be allocated 2 bytes. Assuming that the compiler assigns it memory locations 1003 and 1004, we say the value of x = 10 and the address of x (written as &x) is equal to 1003, i.e.,

the starting address of x in the memory. When we write, `ptr = &x`, then `ptr = 1003`.

> **Note** In C, pointers are not allowed to store memory addresses, but they can store only the addresses of variables of a given type. Therefore the statement `int *ptr = 1000;` is absolutely illegal in C.

Programming Tip: The address of a memory location is a constant, therefore, it cannot be changed in the program code.

We can *dereference* a pointer, i.e., refer to the value of the variable to which it points, by using unary '*' operator (also known as *indirection operator*) as in `*ptr`, i.e., `*ptr = 10`, since `10` is value of x. Therefore, * is equivalent to writing *value at address*. Look at the code below which shows the use of pointer variable.

```
#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);
   printf("\n The address of number in
   memory is: %p", &num);
   return 0;
}
```

Output

```
Enter the number: 10
The number that was entered is: 10
The address of number in memory is: FFDC
```

> **Note** %p control string prints the argument as a memory address in hexadecimal form. %u prints memory address in decimal form.

Programming Tip: A pointer variable cannot be assigned value of the variable to which it points.

Can you tell what will be the value of `*(&num)`? Yes it is equivalent to num. The indirection and the address operators are inverse of each other, so when combined in an expression, they cancel each other.

We can also assign values to variables using pointer variables and modify its value. The code given below shows this.

```
#include <stdio.h>
int main()
{
   int num, *pnum;
   pnum = &num;
   *pnum = 10;
   printf("\n *pnum = %d", *pnum);
   printf("\n num = %d", num);
   *pnum = *pnum + 1;
   // increments the value of num
   printf("\n After increment *pnum = %d",
   *pnum);
   printf("\n After increment num = %d", num);
   return 0;
}
```

Output

```
*pnum = 10
num = 10
After increment *pnum = 11
After increment num = 11
```

Now can you predict the output of the following code?

```
#include <stdio.h>
int main()
{
   int num, *pnum1, *pnum2;
   pnum1 = &num;
   *pnum1 = 10;
   pnum2 = pnum1;
   printf("\n Value of num using all three
   variables (num, *pnum1, *pnum2) = %d %d
   %d", num, *pnum1, *pnum2);
   printf("\n Address of num using all three
   variables (&num, pnum1, pnum2) = %x %x
   %x", num, pnum1, pnum2);
   return 0;
}
```

While the first `printf` statement will print the value of num, the second `printf` statement will print address of num. These are just three different ways to refer to the value and address of the same variable.

> **Note** Any number of pointers can point to the same address.

Programming Tip: It is an error to assign address of a variable to another variable of the same type. The variable to which the adddress is being assigned must be declared as a pointer variable.

The address of a variable is the address of the first byte occupied by that variable. Basically, address of the variable is the relative location of the variable with respect to the program's memory space. We cannot change the address of a variable, we can only use them. Although the address of a variable cannot be changed, the variable's address

may change during different program runs, i.e., if you try to print the address of num today, it may print 4010. Next time when you run the program, it may print the address as FA12.

One thing to remember always is that the data type of the pointer variable and the variable whose address it will store must both be of the same type. Therefore, the following code is not valid.

```
int x = 10;
float y = 2.0;
int *px;
float *py;
px = &y;//INVALID
py = &x;//INVALID
```

Note that it is not necessary that the pointer variable will point to the same variable throughout the program. It can point to any variable as long as the data type of the pointer variable is same as that of the variable it points to. The following code illustrates this concept.

```
#include <stdio.h>
main()
{
   int a = 3, b = 5;
   int *pnum;
   pnum = &a;
   printf("\n a = %d", *pnum);
   pnum = &b;
   printf("\n b = %d", *pnum);
   return 0;
}
```

Output

```
a = 3
b = 5
```

Note: Using an un-initialized pointer can cause unpredictable results.

13.4 POINTER EXPRESSIONS AND POINTER ARITHMETIC

Similar to other variables, pointer variables can also be used in expressions. For example, if ptr1 and ptr2 are pointers, then the following statements are valid.

```
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;
```

Programming Tip: A pointer variable that is declared but not initialied contains garbage value. Hence, such a pointer variable must not be used before it is assigned any variable's address.

In C, the programmer may add or subtract integers from pointers. We can also subtract one pointer from the other. We can also use short hand operators with the pointer variables as we use with other variables.

C also allows to 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 (*). But both these operators have a special behaviour when used as suffix. In that case the expression is evaluated with the value it had before being increased. Therefore, the expression

```
*ptr++
```

is equivalent to *(ptr++) as ++ has greater operator precedence than *. Therefore, the expression will increase the value of ptr so that it now points to the next element. This means the statement *ptr++ does not perform the intended task. Therefore, to increment the value of the variable whose address is stored in ptr, you should write

```
(*ptr)++
```

Now, let us consider another C statement

```
int num1 = 2, num2 = 3;
int *p = &num1, *q = &num2;
*p++ = *q++;
```

What will *p++ = *q++ do? Because ++ has a higher precedence than *, both p and q are increased, but because the increment operators (++) are used as postfix and not prefix, the value assigned to *p is *q before both p and q are increased. Then both are increased. So the statement is equivalent to writing:

Programming Tip: It is an error to subtract two pointer variables.

```
*p = *q;
++p; ++q;
```

Let us now summarize the rules for pointer operations:

- A pointer variable can be assigned the address of another variable (of the same type).
- It can be assigned the value of another pointer variable (of the same type).
- It can be initialized with a NULL (or 0) value.
- Prefix or postfix increment and decrement operators can be applied on a pointer variable.

- An integer value can be added or subtracted from a pointer variable.
- A pointer variable can be compared with another pointer variable of the same type using relational operators.
- A pointer variable cannot be multiplied by a constant.
- A pointer variable cannot be added to another pointer variable.

2. Write a program to print Hello world, using pointers.

```
#include <stdio.h>
int main()
{
   char *ch = "Hello World";
   printf("%s", ch);
   return 0;
}
```

Output

```
Hello World
```

3. Write a program to add two floating point numbers. The result should contain only two digits after the decimal.

```
#include <stdio.h>
int main()
{
   float num1, num2, sum = 0.0;
   float *pnum1 = &num1, *pnum2 = &num2,
   *psum = &sum;
   printf("\n Enter the two numbers: ");
   scanf("%f %f", pnum1, pnum2);
   // pnum1 = &num1;
   *psum = *pnum1 + *pnum2;
   printf("\n %f + %f = %.2f", *pnum1,
   *pnum2, *psum);
   return 0;
}
```

Output

```
Enter the two numbers: 2.5 3.4
2.5 + 3.4 = 5.90
```

4. Write a program to calculate area of a circle.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   double radius, area = 0.0;
   double *pradius = &radius, *parea =
   &area;
   printf("\n Enter the radius of the circle:
   ");
   scanf("%lf", pradius);
   *parea = 3.14 * (*pradius) * (*pradius);
   printf("\n The area of the circle with
   radius %.2lf = %.2lf", *pradius, *parea);
   return 0;
}
```

Output

```
Enter the radius of the circle: 7.0
The area of the circle with radius 7.0 =
  153.83
```

5. Write a program to convert a floating point number into an integer.

```
#include <stdio.h>
int main()
{
   float fnum, *pfnum = &fnum;
   int num, *pnum = &num;
   printf("\n Enter the floating point no.:
   ");
   scanf("%f", &fnum);
   *pnum = (int)*pfnum;
   printf("\n The integer equivalent of %f =
   %d", *pfnum, *pnum);
   return 0;
}
```

Output

```
Enter the floating point no.: 3.4
The integer equivalent of 3.4000 = 3
```

6. Write a program to find the biggest of three numbers.

```
#include <stdio.h>
int main()
{
   int num1, num2, num3;
   int *pnum1 = &num1, *pnum2 = &num2,
   *pnum3= &num3;
   printf("\n Enter the first number: ");
   scanf("%d", pnum1);
   printf("\n Enter the second number: ");
   scanf("%d", pnum2);
   printf("\n Enter the third number: ");
   scanf("%d", pnum3);
   if(*pnum1 > *pnum2 && *pnum1 > *pnum3)
      printf("\n %d is the largest number",
   *pnum1);
   if(*pnum2 > *pnum1 && *pnum2 > *pnum3)
      printf("\n %d is the largest number",
   *pnum2);
   else
      printf("\n %d is the largest number",
   *pnum3);
   return 0;
}
```

Output

```
Enter the first number: 5
Enter the second number: 7
Enter the third number: 3
7 is the largest number
```

7. Write a program to print a character, its ascii value and rewrite the character in upper case.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int ch, *pch = &ch;
   clrscr();
   printf("\n Enter the character: ");
   scanf("%c", &ch);
   printf("\n The char entered is:
   %c", *pch);
   printf("\n ascii value of the char is: %d",
   *pch);
   printf("\n The char in upper case is: %c",
   *pch-32);
   getch();
   return 0;
}
```

Output

```
Enter the character: z
The char entered is: z
ascii value of the char is: 122
The char in upper case is: Z
```

8. Write a program to enter a character and then determine whether it is a vowel or not.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   char ch, *pch = &ch;
   clrscr();
   printf("\n Enter any character: ");
   scanf("%c", pch);
    if(*pch=='a' || *pch=='e' ||
     *pch=='i' || *pch=='o' || *pch=='u' ||
     *pch=='A' || *pch=='E' || *pch=='I'
        || *pch=='O' || *pch=='U')
     printf("\n %c is a VOWEL", ch);
   else
     printf("\n %c is not a vowel", *pch);
   getch();
   return 0;
}
```

Output

```
Enter any character: i
i is a vowel
```

9. Write a program which takes an input from the user and then checks whether it is a number or a character. If it is a character, determine whether it is in upper case or lower case.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   char ch, *pch = &ch;
   clrscr();
   printf("\n Enter any character: ");
   scanf("%c", pch);
   if(*pch >='A' && *pch<='Z')
     printf("\n Upper case char was
   entered");
   if(*pch >='a' && *pch<='z')
     printf("\n Lower case char was
   entered");
   else if(*pch>='0' && *pch<='9')
     printf("\n You entered a number");
   getch();
   return 0;
}
```

Output

```
Enter any character: 7
You entered a number
```

10. Write a program using pointer variables to read a character until * is entered. If the character is in upper case, print it in lower case and vice versa. Also count the number of upper and lower case characters entered.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   char ch, *pch = &ch;
   int upper = 0, lower = 0;
   clrscr();
   printf("\n Enter the character: ");
   scanf("%c", pch);
   while(*pch != '*')
   {
     if(*pch >= 'A' && *pch <= 'Z')
     {
       *pch += 32;
       upper++;
     }
     if(*pch >= 'a' && *pch <= 'z')
     {
       *pch -= 32;
       lower++;
     }
     printf("%c", *pch);
     fflush(stdin);
     printf("\n Enter the character: ");
     scanf("%c", pch);
   }
   printf("\n Total number of upper case
   characters = %d", upper);
   printf("\n Total number of lower case
   characters = %d", lower);
```

```
    getch();
    return 0;
}
```

Output

```
Enter the character: A
A
Enter the character: b
B
Enter the character: c
C
Enter the character: *
Total number of upper case characters = 1
Total number of lower case characters = 2
```

11. Write a program to test whether a number is positive, negative, or equal to zero.

```
#include <stdio.h>
int main()
{
   int num, *pnum = &num;
   printf("\n Enter any number: ");
   scanf("%d", pnum);
   if(*pnum>0)
     printf("\n The number is positive");
   else
   {
     if(*pnum<0)
     printf("\n The number is negative");
     else
     printf("\n The number is equal to
   zero");
   }
   return 0;
}
```

Output

```
Enter any number: -1
The number is negative
```

12. Write a program to display the sum and average of numbers from m to n.

```
#include <stdio.h>
int main()
{
   int num,*pnum = &num, range;
   int m,*pm = &m;
   int n,*pn = &n;
   int sum = 0, *psum = &sum;
   float avg, *pavg = &avg;
   printf("\n Enter the starting and ending
   limit of the numbers to be summed:");
      scanf("%d %d", pm, pn);
      range = n - m ;
   while(*pm <= *pn)
   {
     *psum = *psum + *pm;
      *pm = *pm + 1;
   }
   printf("\n Sum of numbers = %d , *psum);
   *pavg = *psum / range;
   printf("\n Average of numbers = %f",
   *pavg);
   return 0;
}
```

Output

```
Enter the starting and ending limit of
   the numbers to be summed: 0 10
Sum of numbers = 55
Average of numbers = 5.5
```

13. Write a program to print all even numbers from m to n.

```
#include <stdio.h>
#include <conio.h>
int main()
{
   int m, *pm = &m;
   int n, *pn = &n;
   printf("\n Enter the starting and ending
   limit of the numbers: ");
      scanf("%d %d", pm, pn);
   while(*pm <= *pn)
   {
      if(*pm %2 == 0)
      printf("\n %d is even", *pm);
      else
      printf("\n %d is odd", *pm);
   }
   return 0;
}
```

Output

```
Enter the starting and ending limit of
   the numbers: 0 10
0 is even
2 is even
4 is even
6 is even
8 is even
10 is even
```

14. Write a program to read numbers until –1 is entered and display whether the number is prime or composite.

```
#include <stdio.h>
int main()
{
   int num, *pnum = &num;
   int i, flag = 0;
   printf("\n ***** ENTER -1 TO EXIT
   ******");
   printf("\n Enter any number: ");
   scanf("%d", pnum);
   while(*pnum != -1)
```

```
    {
       if(*pnum == 1)
       printf("\n %d is neither prime nor
    composite", *pnum);
       else if(*pnum == 2)
       printf("\n %d is prime", *pnum);
       else
       {
       for(i=2; i<*pnum/2; i++)
       {
       if(*pnum/i == 0)
       flag =1;
       }
       if(flag == 0)
       printf("\n %d is prime", *pnum);
       else
       printf("\n %d is composite", *pnum);
       }
       printf("\n Enter any number: ");
       scanf("%d", pnum);
    }
    return 0;
}
```

Output

```
***** ENTER -1 TO EXIT ******
Enter any number: 3
3 is prime
Enter any number: 1
1 is neither prime nor composite
Enter any number: -1
```

13.5 NULL POINTERS

We have seen that the pointer variable is a pointer to some other variable of the same data type. However, in some cases we may prefer to have *null pointer* which is a special pointer value that does not 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, which is defined in several standard header files including <stdio.h>, <stdlib.h>, and <string.h>. After including any of these files in your program just write,

```
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;
}
```

You may also initialize a pointer as a null pointer by using a constant 0, as follows:

```
int ptr,
ptr = 0;
```

Programming Tip: It is a logical error to dereference a null pointer.

This is a valid statement in C, as even NULL which is a preprocessor macro typically has the value, or replacement text, 0. However, to avoid ambiguity it is always better to use NULL to declare a null pointer.

A function that returns pointer values can return a null pointer when it is unable to perform its task.

Null pointers are used in situations where one of the pointers in the program points to different locations at different times. 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.

Note A run-time error is generated if you try to dereference a null pointer.

13.6 GENERIC POINTERS

A *generic pointer* is a pointer variable that has void as its data type. The void pointer, or the generic pointer, is a special type of pointer that can be used to point to variables of any data type. It is declared like a normal pointer variable but using the void keyword as the pointer's data type. For example,

Programming Tip: A compiler error will be generated if you assign a pointer of one type to a pointer of another type without a cast.

```
void *ptr;
```

In C, since you cannot have a variable of type void, the void pointer will, therefore, not point to any data and thus, cannot be dereferenced. You need to type cast a void pointer (generic pointer) to another kind of pointer before using it.

Programming Tip: A pointer variable of type void * cannot be dereferenced.

Generic pointers are often used when you want a pointer to point to data of different types at different times. For example, look at the code given below.
Here is some code using a void pointer:

```
#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
```

It is always recommended to avoid using void pointers unless absolutely necessary, as they effectively allow you to avoid type checking.

13.7 PASSING ARGUMENTS TO FUNCTIONS USING POINTERS

We have already seen call by value method of passing parameters to a function. Using call by value method, it is impossible to modify the actual parameters in the call when you pass them to a function. Furthermore, the incoming arguments to a function are treated as local variables in the function and those local variables get a *copy* of the values passed from their caller.

Pointers provide a mechanism to modify data declared in one function using code written in another function. In other words: if data is declared in `func1()` and we want to write code in `func2()` that modifies the data in `func1()`, then we must pass the addresses of the variables we want to change.

Programming Tip: While using pointers to pass arguments to a function, the calling function must pass addresses of the variables as arguments and the called function must dereference the arguments to use them in the function body for processing.

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 avoids the overhead of copying data from one function to another. Hence, to use pointers for passing arguments to a function, the programmer must do the following:

- Declare the function parameters as pointers
- Use the dereferenced pointers in the function body
- Pass the addresses as the actual argument when the function is called.

Note It is an error to return a pointer to a local variable in the called function, because when the function terminates, its memory may be allocated to a different program.

Let us write some programs that pass pointer variables as parameters to functions.

15. Write a program to add two integers using functions

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

Output

```
Enter the first number: 2
Enter the first number: 3
Total = 5
```