POINTERS AND CHARACTER STRINGS

We have seen in Chapter 8 that strings are treated like character arrays and therefore, they are declared and initialized as follows:

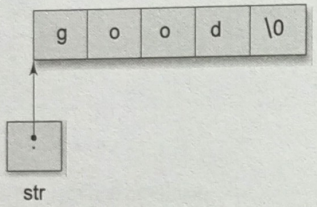
```
char str [5] = "good";
```

The compiler automatically inserts the null character '\0' at the end of the string. C supports an alternative method to create strings using pointer variables of type **char**. Example:

```
char *str = "good";
```

This creates a string for the literal and then stores its address in the pointer variable **str**.

The pointer **str** now points to the first character of the string "good" as:



We can also use the run-time assignment for giving values to a string pointer. Example

```
char * string1;
string1 = "good";
```

Note that the assignment

```
string1 = "good";
```

is not a string copy, because the variable **string1** is a pointer, not a string.

(As pointed out in Chapter 8, C does not support copying one string to another through the assignment operation.)

We can print the content of the string **string1** using either **printf** or **puts** functions as follows:

```
printf("%s", string1);
puts (string1);
```

Remember, although **string1** is a pointer to the string, it is also the name of the string. Therefore, we do not need to use indirection operator * here.

Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated by the Worked-Out Problem 11.5.

WORKED-OUT PROBLEM 11.5

E

Write a program using pointers to determine the length of a character string.

A program to count the length of a string is shown in Fig.11.10. The statement

```
char *cptr = name;
```

declares **cptr** as a pointer to a character and assigns the address of the first character of **name** as the initial value. Since a string is always terminated by the null character, the statement

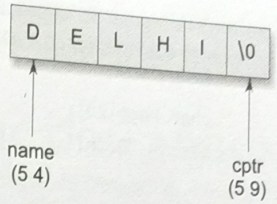
```
while(*cptr != '\0')
```

is true until the end of the string is reached.

When the **while** loop is terminated, the pointer **cptr** holds the address of the null character. Therefore, the statement

```
length = cptr - name;
```

gives the length of the string **name**.



each character stored in 1 byte memory a Each 1 byte memory has address & storing is contiguous

The output also shows the address location of each character. Note that each character occupies one memory cell (byte).

Program

```
main()
{
    char  *name;
    int   length;
    char  *cptr = name;
    name  = "DELHI";
    printf ("%s\n", name);
    while(*cptr != '\0')
    {
        printf("%c is stored at address %u\n", *cptr, cptr);
        cptr++;
    }
    length = cptr - name;
    printf("\nLength of the string = %d\n", length);
}
```

Output

```
DELHI
D is stored at address 54
E is stored at address 55
L is stored at address 56
H is stored at address 57
I is stored at address 58

Length of the string = 5
```

Fig. 11.10 *String handling by pointers*

In C, a constant character string always represents a pointer to that string. And therefore the following statements are valid:

```
char *name;
name = "Delhi";
```

These statements will declare **name** as a pointer to character and assign to **name** the constant character string "Delhi". You might remember that this type of assignment does not apply to character arrays. The statements like

```
char name[20];
name = "Delhi";
```

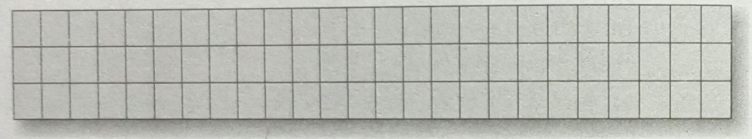
do not work.

ARRAY OF POINTERS

One important use of pointers is in handling of a table of strings. Consider the following array of strings:

```
char name [3][25];
```

This says that the **name** is a table containing three names, each with a maximum length of 25 characters (including null character). The total storage requirements for the **name** table are 75 bytes.



We know that rarely the individual strings will be of equal lengths. Therefore, instead of making each row a fixed number of characters, we can make it a pointer to a string of varying length. For example,

```
char *name[3] = {
                    "New Zealand",
                    "Australia",
                    "India"
                };
```

declares **name** to be an *array of three pointers* to characters, each pointer pointing to a particular name as:

name [0] ⟶ New Zealand
name [1] ⟶ Australia
name [2] ⟶ India

This declaration allocates only 28 bytes, sufficient to hold all the characters as shown

N	e	w		Z	e	a	l	a	n	d	\0
A	u	s	t	r	a	l	i	a	\0		
I	n	d	i	a	\0						

The following statement would print out all the three names:

```
for(i = 0; i <= 2; i++)
    printf("%s\n", name[i]);
```

To access the jth character in the ith name, we may write as

```
*(name[i]+j)
```

The character arrays with the rows of varying length are called 'ragged arrays' and are better handled by pointers.

Remember the difference between the notations ***p[3]** and **(*p)[3]**. Since * has a lower precedence than [ ], *p[3] declares p as an array of 3 pointers while **(*p)[3]** declares **p** as a pointer to an array of three elements.

POINTERS AS FUNCTION ARGUMENTS

LO 11.6
Explain how pointers are used with functions and structures

We have seen earlier that when an array is passed to a function as an argument, only the address of the first element of the array is passed, but not the actual values of the array elements. If **x** is an array, when we call **sort(x)**, the address of **x[0]** is passed to the function **sort**. The function uses this address for manipulating the array elements. Similarly, we can pass the address of a variable as an argument to a function in the normal fashion. We used this method when discussing functions that return multiple values (see Chapter 9).

When we pass addresses to a function, the parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variables is known as *'call by reference'*. (You know, the process of passing the actual value of variables is known as "call by value".) The function which is called by 'reference' can change the value of the variable used in the call.

Consider the following code:

```
main()
{
        int x;
        x = 20;
        change(&x);  /* call by reference or address */
        printf("%d\n",x);
}
change(int *p)
{
        *p = *p + 10;
}
```

When the function **change( )** is called, the address of the variable **x**, not its value, is passed into the function **change()**. Inside **change( )**, the variable **p** is declared as a pointer and therefore **p** is the address of the variable **x**. The statement,

```
*p = *p + 10;
```

means 'add 10 to the value stored at the address **p**'. Since **p** represents the address of **x**, the value of **x** is changed from 20 to 30. Therefore, the output of the program will be 30, not 20.

Thus, call by reference provides a mechanism by which the function can change the stored values in the calling function. Note that this mechanism is also known as "*call by address*" or "*pass by pointers*".

Note *C99 adds a new qualifier* ***restrict*** *to the pointers passed as function parameters. See the Appendix "C99 Features".*

M

WORKED-OUT PROBLEM 11.6

Write a function using pointers to exchange the values stored in two locations in the memory.

The program in Fig. 11.11 shows how the contents of two locations can be exchanged using their address locations. The function **exchange()** receives the addresses of the variables **x** and **y** and exchanges their contents.

Program

```
void exchange (int *, int *);    /* prototype */
main()
{
     int  x, y;
     x = 100;
     y = 200;
     printf("Before exchange  : x = %d   y = %d\n\n", x, y);
     exchange(&x,&y);      /* call */
     printf("After exchange   : x = %d   y = %d\n\n", x, y);
}
exchange (int *a, int *b)
{
     int t;
     t = *a;    /* Assign the value at address a to t */
     *a = *b;   /* put b into a */
     *b = t;    /* put t into b */
}
```

Output

```
Before exchange  : x = 100   y = 200
After exchange   : x = 200   y = 100
```

Fig. 11.11 *Passing of pointers as function parameters*

You may note the following points:

1. The function parameters are declared as pointers.
2. The dereferenced pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments.

The use of pointers to access array elements is very common in C. We have used a pointer to traverse array elements in Program 11.4. We can also use this technique in designing user-defined functions discussed in Chapter 9. Let us consider the problem sorting an array of integers discussed in Program 9.6.

The function **sort** may be written using pointers (instead of array indexing) as shown:

```
void sort (int m, int *x)
{    int i j, temp;
     for (i=1; i<= m-1; i++)
          for (j=1; j<= m-1; j++)
          if (*(x+j-1) >= *(x+j))
          {
               temp = *(x+j-1);
               *(x+j-1) = *(x+j);
               *(x+j) = temp;
          }
}
```


Note that we have used the pointer x (instead of array x[ ]) to receive the address of array passed and therefore the pointer x can be used to access the array elements (as pointed out in Section 11.10). This function can be used to sort an array of integers as follows:

```
     . . . . .
int score[4] = {45, 90, 71, 83};
     . . . . .
sort(4, score); /* Function call */
     . . . . .
```

The calling function must use the following prototype declaration.

```
void sort (int, int *);
```

This tells the compiler that the formal argument that receives the array is a pointer, not array variable.

Pointer parameters are commonly employed in string functions. Consider the function copy which copies one string to another.

```
copy(char *s1, char *s2)
{
     while( (*s1++ = *s2++) != '\0')
     ;
}
```

This copies the contents of **s2** into the string **s1.** Parameters **s1** and **s2** are the pointers to character strings, whose initial values are passed from the calling function. For example, the calling statement

```
copy(name1, name2);
```

will assign the address of the first element of **name1** to **s1** and the address of the first element of **name2** to **s2.**

Note that the value of ***s2++** is the character that **s2** pointed to before **s2** was incremented. Due to the postfix ++, **s2** is incremented only after the current value has been fetched. Similarly, **s1** is incremented only after the assignment has been completed.

Each character, after it has been copied, is compared with '\0' and therefore, copying is terminated as soon as the '\0' is copied.

M

WORKED-OUT PROBLEM 11.7

The program of Fig. 11.12 shows how to calculate the sum of two numbers which are passed as arguments using the call by reference method.

Program

```
#include<stdio.h>
#include<conio.h>
void swap (int *p, *q);
main()
{
```

```
    int x=0;
    int y=20;
    clrstr();
    printf("\nValue of X and Y before swapping are X=%d and Y=%d", x,y);
    swap(&x, &y);
    printf("\n\nValue of X and Y after swapping are X=%d and Y=%d", x,y);
    getch();
    }
    void swap(int *p, int *q)//Value of x and y are transferred using call by
    reference
    {
    int r;
    r=*p;
    *p=*q;
    *q=r;
    }
Output
    Value of X and Y before swapping are X=10 and Y=20
    Value of X and Y after swapping are X=20 and Y=10
```

Fig. 11.12 *Program to pass the arguments using call by reference method*

FUNCTIONS RETURNING POINTERS

We have seen so far that a function can return a single value by its name or return multiple values through pointer parameters. Since pointers are a data type in C, we can also force a function to return a pointer to the calling function. Consider the following code:

```
int *larger (int *, int *);      /* prototype */
main ( )
{
        int a = 10;
        int b = 20;
        int *p;
        p = larger(&a, &b);  /Function call */
        printf ("%d", *p);
}
int *larger (int *x, int *y)
{
    if (*x>*y)
        return (x);  / *address of a */
    else
        return (y);  /* address of b */
}
```


The function **larger** receives the addresses of the variables **a** and **b**, decides which one is larger using the pointers **x** and **y** and then returns the address of its location. The returned value is then assigned to the pointer variable **p** in the calling function. In this case, the address of **b** is returned and assigned to **p** and therefore the output will be the value of **b**, namely, 20.

Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

POINTERS TO FUNCTIONS

A function, like a variable, has a type and an address location in the memory. It is therefore, possible to declare a pointer to a function, which can then be used as an argument in another function. A pointer to a function is declared as follows:

```
type (*fptr) ();
```

This tells the compiler that **fptr** is a pointer to a function, which returns *type* value. The parentheses around ***fptr** are necessary. Remember that a statement like

```
type *gptr();
```

would declare **gptr** as a function returning a pointer to *type*.

We can make a function pointer to point to a specific function by simply assigning the name of the function to the pointer. For example, the statements

```
double mul(int, int);
double (*p1)();
p1 = mul;
```

declare **p1** as a pointer to a function and **mul** as a function and then make **p1** to point to the function **mul**. To call the function **mul**, we may now use the pointer **p1** with the list of parameters. That is,

```
(*p1)(x,y) /* Function call */
```

is equivalent to

```
mul(x,y)
```

Note the parentheses around ***p1**.

WORKED-OUT PROBLEM 11.8 H

Write a program that uses a function pointer as a function argument.

A program to print the function values over a given range of values is shown in Fig. 11.13. The printing is done by the function **table** by evaluating the function passed to it by the **main.**

With **table,** we declare the parameter **f** as a pointer to a function as follows:

```
double (*f)();
```

The value returned by the function is of type **double.** When **table** is called in the statement

```
table (y, 0.0, 2, 0.5);
```

we pass a pointer to the function **y** as the first parameter of **table**. Note that **y** is not followed by a parameter list.

During the execution of **table,** the statement

```
value = (*f)(a);
```

calls the function **y** which is pointed to by **f,** passing it the parameter **a.** Thus the function **y** is evaluated over the range 0.0 to 2.0 at the intervals of 0.5.

Similarly, the call

```
table (cos, 0.0, PI, 0.5);
```

passes a pointer to **cos** as its first parameter and therefore, the function **table** evaluates the value of **cos** over the range 0.0 to PI at the intervals of 0.5.

Program

```
#include  <math.h>
#define  PI  3.1415926
double y(double);
double cos(double);
double table (double(*f)(), double, double, double);

main()
{  printf("Table of y(x) = 2*x*x-x+1\n\n");
   table(y, 0.0, 2.0, 0.5);
   printf("\nTable of cos(x)\n\n");
   table(cos, 0.0, PI, 0.5);
}
double table(double(*f)(),double min, double max, double step)
{     double a, value;
      for(a = min; a <= max; a += step)
      {
         value = (*f)(a);
         printf("%5.2f  %10.4f\n", a, value);
      }
}
double y(double x)
{
return(2*x*x-x+1);
}
```

Output

```
Table of y(x) = 2*x*x-x+1
   0.00       1.0000
   0.50       1.0000
   1.00       2.0000
   1.50       4.0000
   2.00       7.0000
Table of cos(x)
   0.00       1.0000
   0.50       0.8776
   1.00       0.5403
   1.50       0.0707
   2.00      -0.4161
   2.50      -0.8011
   3.00      -0.9900
```

Fig. 11.13 *Use of pointers to functions*

Compatibility and Casting

A variable declared as a pointer is not just a *pointer type* variable. It is also a pointer to a *specific* fundamental data type, such as a character. A pointer therefore always has a type associated with it. We cannot assign a pointer of one type to a pointer of another type, although both of them have memory addresses as their values. This is known as *incompatibility* of pointers.

All the pointer variables store memory addresses, which are compatible, but what is not compatible is the underlying data type to which they point to. We cannot use the assignment operator with the pointers of different types. We can however make explicit assignment between incompatible pointer types by using **cast** operator, as we do with the fundamental types. Example:

```
int x;
char *p;
p = (char *) & x;
```

In such cases, we must ensure that all operations that use the pointer **p** must apply casting properly.

We have an exception. The exception is the void pointer (void *). The void pointer is a ***generic pointer*** that can represent any pointer type. All pointer types can be assigned to a void pointer and a void pointer can be assigned to any pointer without casting. A void pointer is created as follows:

```
void *vp;
```

Remember that since a void pointer has no object type, it cannot be de-referenced.

POINTERS AND STRUCTURES

We know that the name of an array stands for the address of its zeroth element. The same thing is true of the names of arrays of structure variables. Suppose **product** is an array variable of **struct** type. The name **product** represents the address of its zeroth element. Consider the following declaration:

```
struct inventory
{
      char    name[30];
      int     number;
      float   price;
}     product[2], *ptr;
```

This statement declares **product** as an array of two elements, each of the type **struct inventory** and **ptr** as a pointer to data objects of the type **struct inventory**. The assignment

```
ptr = product;
```

would assign the address of the zeroth element of **product** to **ptr**. That is, the pointer **ptr** will now point to **product[0]**. Its members can be accessed using the following notation.

```
ptr -> name
ptr -> number
ptr -> price
```

The symbol –> is called the *arrow operator* (also known as *member selection operator*) and is made up of a minus sign and a greater than sign. Note that **ptr–>** is simply another way of writing **product[0].** When the pointer **ptr** is incremented by one, it is made to point to the next record, i.e., product[1]. The following **for** statement will print the values of members of all the elements of **product** array.

```
for(ptr = product; ptr < product+2; ptr++)
      printf ("%s %d %f\n", ptr->name, ptr->number, ptr->price);
```

We could also use the notation

```
(*ptr).number
```

to access the member **number**. The parentheses around ***ptr** are necessary because the member operator '.' has a higher precedence than the operator *.

WORKED-OUT PROBLEM 11.9

H

Write a program to illustrate the use of structure pointers.

A program to illustrate the use of a structure pointer to manipulate the elements of an array of structures is shown in Fig. 11.14. The program highlights all the features discussed above. Note that the pointer **ptr** (of type **struct invent**) is also used as the loop control index in **for** loops.

Program

```
struct invent
{
   char    *name[20];
   int     number;
   float price;
};
main()
{
   struct invent product[3], *ptr;
   printf("INPUT\n\n");
   for(ptr = product; ptr < product+3; ptr++)
      scanf("%s %d %f", ptr->name, &ptr->number, &ptr->price);
   printf("\nOUTPUT\n\n");
      ptr = product;
      while(ptr < product + 3)
      {
           printf("%-20s %5d %10.2f\n",
                  ptr->name,
                  ptr->number,
                  ptr->price);
           ptr++;
      }
   }
```

Output

```
INPUT
Washing_machine    5     7500
Electric_iron      12    350
Two_in_one         7     1250

OUTPUT
Washing machine    5     7500.00
Electric_iron      12    350.00
Two_in_one         7     1250.00
```


Fig. 11.14 *Pointer to structure variables*

While using structure pointers, we should take care of the precedence of operators.

The operators '–>' and '.', and () and [] enjoy the highest priority among the operators. They bind very tightly with their operands. For example, given the definition

```
struct
{
   int count;
   float *p;      /* pointer inside the struct */
} ptr;            /* struct type pointer */
```

then the statement

```
++ptr->count;
```

increments **count**, not **ptr**. However,

```
(++ptr)->count;
```

increments **ptr** first, and then links **count**. The statement

```
ptr++ -> count;
```

is legal and increments **ptr** after accessing **count**.

The following statements also behave in the similar fashion.

***ptr–>p**	Fetches whatever **p** points to.
***ptr–>p++**	Increments **p** after accessing whatever it points to.
(*ptr–>p)++	Increments whatever **p** points to.
***ptr++–>p**	Increments **ptr** after accessing whatever it points to.

In the previous chapter, we discussed about passing of a structure as an argument to a function. We also saw an example where a function receives a copy of an entire structure and returns it after working on it. As we mentioned earlier, this method is inefficient in terms of both, the execution speed and memory. We can overcome this drawback by passing a pointer to the structure and then using this pointer to work on the structure members. Consider the following function:

```
print_invent(struct invent *item)
   {
      printf("Name: %s\n", item->name);
      printf("Price: %f\n", item->price);
   }
```

This function can be called by

```
print_invent(&product);
```

The formal argument **item** receives the address of the structure **product** and therefore it must be declared as a pointer of type **struct invent**, which represents the structure of **product**.

TROUBLES WITH POINTERS

Pointers give us tremendous power and flexibility. However, they could become a nightmare when they are not used correctly. The major problem with wrong use of pointers is that the compiler may not detect the error in most cases and therefore the program is likely to produce unexpected results. The output may not given us any clue regarding the use of a bad pointer. Debugging therefore becomes a difficult task.

We list here some pointer errors that are more commonly committed by the programmers.

- Assigning values to uninitialized pointers

```
int * p, m = 100 ;
*p = m ;              /* Error */
```

- Assigning value to a pointer variable

```
int *p, m = 100 ;
p = m;                /* Error */
```

- Not dereferencing a pointer when required

```
int *p, x = 100;
p = &x;
printf("%d",p);     /* Error */
```

- Assigning the address of an uninitialized variable

```
int m, *p
p = &m;               /* Error */
```

- Comparing pointers that point to different objects

```
char name1 [ 20 ], name2 [ 30 ];
char *p1 = name1;
char *p2 = name2;
if(p1 > p2).....          /* Error */
```

We must be careful in declaring and assigning values to pointers correctly before using them. We must also make sure that we apply the address operator & and referencing operator * correctly to the pointers. That will save us from sleepless nights.

KEY CONCEPTS

- **MEMORY:** This is a sequential collection of storage cells with each cell having an address value associated with it. **[LO 11.1]**
- **POINTER:** It is used to store the memory address as value. **[LO 11.1]**
- **POINTER VARIABLE:** It is a variable that stores the memory address of another variable. **[LO 11.2]**
- **CALL BY REFERENCE:** It is the process of calling a function using pointers to pass the addresses of variables. **[LO 11.6]**
- **CALL BY VALUE:** It is the process of passing the actual value of variables. **[LO 11.6]**

One of the most common application areas of pointers and structures is in the implementation of linear data structures, such as stacks, queues and linked lists. Scan the QR Code

OR visit the link
http://qrcode.flipick.com/index.php/370
to learn how pointers and structures are used for implementing stacks in C