5

Pointers

CHAPTER OUTLINE

5.1 INTRODUCTION

Pointers are the most powerful feature of 'C'. The beginners of 'C' find pointers hard to understand and manipulate. In an attempt to unveil the mystery behind this aspect, the basics of pointers which generally remain in background are being discussed in the following sections.

5.1.1 The '&' Operator

When a variable x is declared in a program, a storage location in the main memory is made available by the compiler. For example, Figure 5.1 shows the declaration of x as an integer variable and its storage location in the main memory.

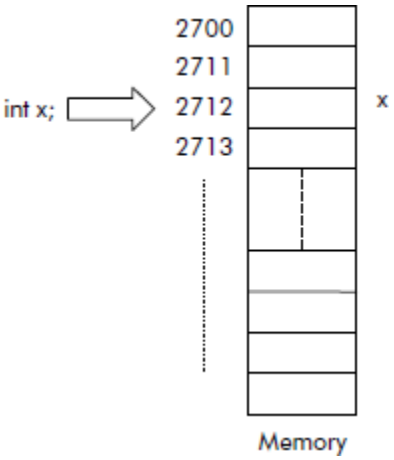


Fig. 5.1 The variable x and its equivalent representation in memory

From Figure 5.1, it may be observed that x is the name associated by the compiler to a location in the memory of the computer. Let us assume that, at the

time of execution, the physical address of this memory location (called `x`) `is` 2712. Now, a point worth noting is that this memory location is viewed by the programmer as variable x and by the operating system as an address 2712. The address of variable x can be obtained by `'&'` an address of operator. This operator when applied to a variable gives the physical memory address of the variable. Thus, `&x` will provide the address 2712. Consider the following program segment:

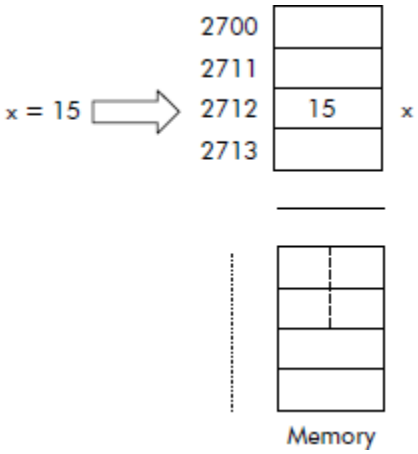


Fig. 5.2 The contents of variable x

```
x=15;

printf('\n value of x = %d', x);

printf('\n address of x = %d', &x);
```

The output of the above program segment would be as shown below:

```
Value of x = 15

Address of x = 2712 (assumed value)
```

The contents of variable x (memory location 2712) are shown in Figure 5.2.

It may be noted here that the value of address of x (i.e., 2712) is assumed and the actual value is dependent on '*machine and execution*' time.

5.1.2 The '*' Operator

The '*' is an indirection operator or `'value at address operator'`. In simple words, we can say that if address of a variable is known, then the '*'

operator provides the contents of the variable. Consider the following program segment:

```
#include <stdio.h>

main()

{

  int x =15;

  printf ("\n Value of x = %d", x);

  printf ("\n Address of x = %u", &x);

  printf ("\n Value at address %d = %d", *(&x));

}
```

The output of the above program segment would be as shown below:

```
  Value of x = 15

  Address of x = 2712

  Value at address 2712 = 15
```

5.2 POINTER VARIABLES

An address of a variable can be stored in a special variable called pointer variable. A pointer variable contains the address of another variable. Let us assume that `y` is such a variable. Now, the address of variable `x` can be assigned to `y` by the statement: `y = &x.` The effect of this statement is shown in Figure 5.3.

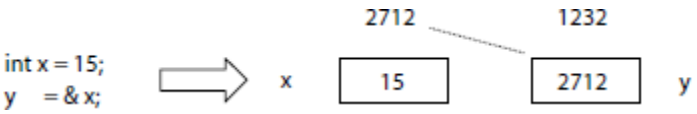


Fig. 5.3 The address of x is being stored in pointer variable y

From Figure 5.3, it is clear that `y` is the variable that contains the address (i.e., 2712) of another variable `x,` whereas the address of `y` itself is 1232 (assumed value). In other words, we can say that y is a pointer to variable `x.` See Figure 5.4.

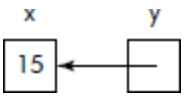
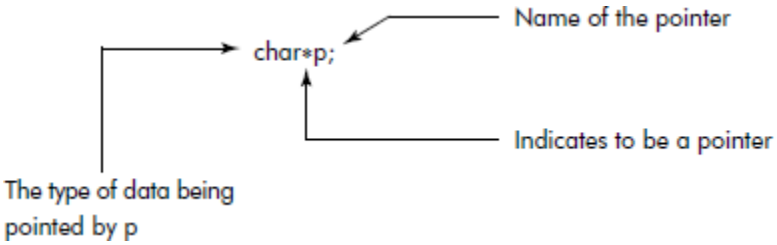


Fig. 5.4 y is a pointer to x

A pointer variable `y` can be declared in 'C' as shown below:

```
int *y;
```

The above declaration means that `y` is a pointer to a variable of type `int`. Similarly, consider the following declaration:



The above declaration means that `p` is a pointer to a variable of type `char`.

Consider the following program segment:

```
#include <stdio.h>

main()

{

  int x = 15;

  int *y;

  y = &x;

  printf ("\n Value of x = %d", x);

  printf ("\n Address of x = %u", &x);

  printf ("\n Value of x = %d", *y);

  printf ("\n Address of x = %u", y);
```

```
  printf ("\n Address of y = %u", &y);
}
```

The output of the above program segment would be as shown below:

```
  Value of x = 15
  Address of x = 2712
  Value of x = 15
  Address of x = 2712
  Address of y = 1232
```

Let us, now, consider the following statements:

```
  float val = 35.67;
  float *pt;
```

The first statement declares `val` to be a variable of type `float` and initializes it by value 35.67. The second statement declares pt to be pointer to a variable of type `float`. Please notice that `pt` can point to a variable of type `float` but it is currently not pointing to any variable as shown in Figure 5.5.

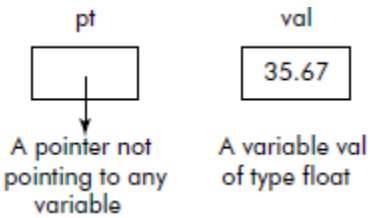


Fig. 5.5 The pointer pt is not pointing to any variable

The pointer `pt` can point to the variable `val` by assigning the address of `val` to `pt` as shown below:

```
  pt = &val;
```

Once the above statement is executed, the pointer `pt` gets the address of `val` and we can say that logically, `pt` points to the variable `val` as shown in Figure 5.6.

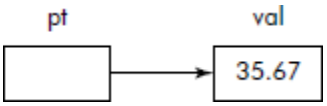


Fig. 5.6 The pointer pt pointing to variable val

However, one must keep in mind that the arrow is for illustration `sake.` Actually, the pointer pt contains the address of `val.` The address is an integer which is not a simple value but a reference to a location in the memory.

Examples of some valid pointer declarations are:

```
(i) int *p;

(ii) char *i, *k;

(iii) float *pt;

(iv) int **ptr;
```

The example (iv) means the ptr is a pointer to a location which itself is a pointer to a variable of type integer.

It may be noted here that a pointer can also be incremented to point to an immediately next location of its type. For example, if the contents of a pointer p of type integer are 5224 then the content of `p++` will be 5226 instead of 5225 (see Figure 5.7). The reason being that an int is always of 2 bytes size and, therefore, stored in two memory locations as shown in Figure 5.7

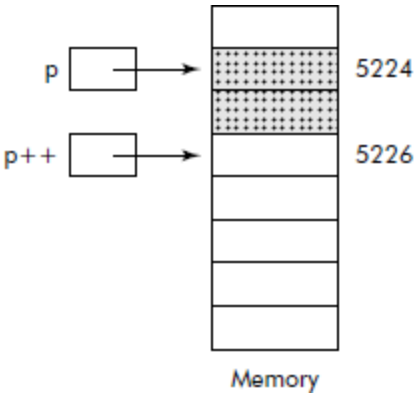


Fig. 5.7 The variable of type int occupies two bytes

The amount of storage taken by various types of data is tabulated in Table 5.1.

Table 5.1 Amount of storage taken by data types

Data type	Amount of storage
character	1 byte
integer	2 byte
float	4 byte
Rong	4 byte
double	8 byte

Thus, a pointer of float type will point to an address of 4 bytes of location and, therefore, an increment to this pointer will increment its contents by 4 locations. It may be further noted that more than one pointer can point to the same location. Consider the following program segment:

```
int x = 37

int *p1, *p2;
```

The following statement:

```
p1 = &x;
```

Makes the pointer `p1` point to variable `x` as shown in Figure 5.8.

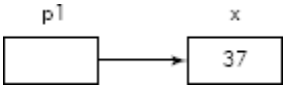


Fig. 5.8 Pointer p1 points to variable x

The following statement:

```
p2 = p1
```

Makes `p2` point to the same location which is being pointed by `p1` as shown in Figure 5.9.

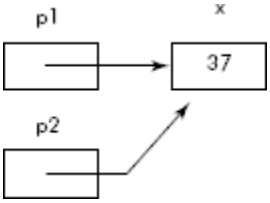


Fig. 5.9 Pointers p1 and p2 pointing to x

The contents of variable `x` are now reachable by both the pointers `p1 and p2`. Thus, two or more entities can cooperatively share a single memory structure with the help of pointers. In fact, this technique can be used to provide efficient communication between different parts of a program.

The pointers can be used for a variety of purposes in a program; some of them are as follows:

- To pass address of variables from one function to another.
- To return more than one value from a function to the calling function.
- For the creation of linked structures such as linked lists and trees.

Example 1: Write a program that reads from the keyboard two variables `x and y` of type integer. It prints the exchanged contents of these variables with the help of pointers without altering the variables.

Solution: We will use two pointers `p1 and p2` for variables `x and y.` A third pointer `p3` will be used as a temporary pointer for the exchange of pointers `p1 and p2.` The scheme is shown in Figure 5.10.

The program is given below:

```
/* This program illustrates the usage of pointers to
exchange the contents of two variables */

#include <stdio.h>

main()

{

 int x, y;

 int *p1, *p2, *p3; /* pointers to integers */

 printf ("\n Enter two integer values");

 scanf ("%d %d", &x, &y);

   /* Assign the addresses x and y to p1 and p2 */

 p1 = &x;

 p2 = &y;

   /* Exchange the pointers */

 p3 = p1;

 p1 = p2;

 p2 = p3;

   /* Print the contents through exchanged contents */

 printf ("\n The exchanged contents are");

 printf (" %d & %d", *p1, *p2);
```

```
}
```

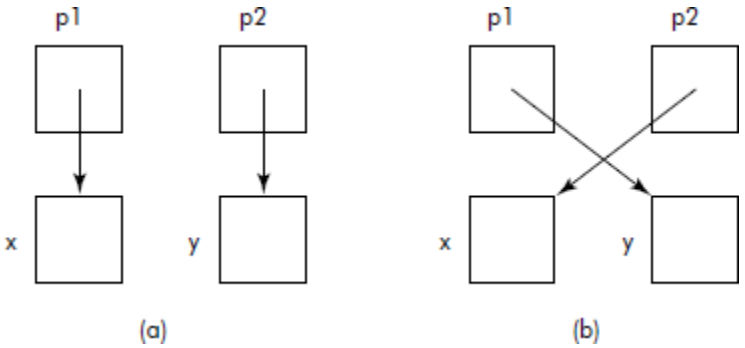


Fig. 5.10 Exchange of contents of variables by exchanging pointers

Note: The output would show the exchanged values because of exchange of pointers (see Figure 5.10) whereas the contents of `x and y` have remained unaltered.

5.2.1 Dangling pointers

A dangling pointer is a pointer which has been allocated but does not point to any entity. Such an un-initialized pointer is dangerous in the sense that a dereference operation on it will result in an unpredictable operation or, may be, a runtime error.

```
int * ptr;

*ptr = 50;
```

The first statement allocates the pointer called `ptr` as shown in Figure 5.11. It may be noted that, currently, `ptr` is a dangling pointer. The second statement is trying to load a value (i.e., 50) to a location which is pointed by ptr. Obviously, this is a dangerous situation because the results will be unpredictable. The 'C' compiler does not consider it as an error, though some compilers may give a warning.

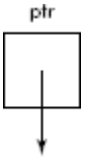


Fig. 5.11 The dangling pointer

Therefore, following steps must be followed if pointers are being used:

Steps:

1. Allocate the pointer.
2. Allocate the variable or entity to which the pointer is to be pointed.
3. Point the pointer to the entity.

Let us assume that it is desired that the pointer `ptr` should point to a variable `val` of type `int`. The correct code in that case would be as given below:

```
int * ptr;

int val;

ptr = &val;

*ptr = 50;
```

The result of the above program segment is shown in Figure 5.12.

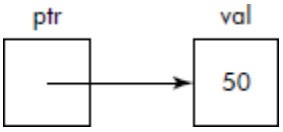


Fig. 5.12 The correct assignment

5.3 POINTERS AND ARRAYS

Pointers and arrays are very closely related to each other. In 'C', the name of an array is a pointer that contains the base address of the array. In fact, it is a pointer to the first element of the array. Consider the array declaration given below:

```
int list[] = {20, 30, 35, 36, 39};
```

This array will be stored in the contiguous locations of the main memory with starting address equal to 1001(assumed value), as shown in Figure 5.13. Since the array called 'list' is of integer type, each element of this array occupies two bytes. The name of the array contains the starting address of the array, i.e., 1001.

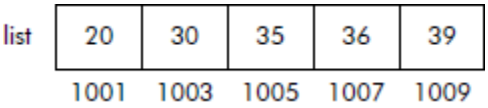


Fig. 5.13 The contiguous storage allocation with starting address = 1001

Consider the following program segment:

```
 printf ("\n Address of zeroth element of array list=
%d", list);

 printf ("\n Value of zeroth element of array list=
%d", *list);
```

Once the above program segment is executed, the output would be as shown below:

```
 Address of zeroth element of array list = 1001

 Value of zeroth element of array list = 20
```

We could have achieved the same output by the following program segment also:

```
 printf ("\n Address of zeroth element of array list =
%d", & list [0]);

 printf ("\n Value of zeroth element of array list =
%d", list []);
```

Both the above given approaches are equivalent because of the following equivalence relations:

```
 list ≡ &list [0] - both denote address of zeroth
element of the array list

 *list ≡ list[0] - both denote value of zeroth element
of the array list
```

The left side approach is known as pointer method and right side as array indexing method. Let us now write a program that prints out a list by array indexing method.

```
/* Array indexing method */

#include <stdio.h>

main()

{
```

```
  static int list [] = {20, 30, 35, 36, 39};
  int i;
  printf ("\n The list is …");
  for (i = 0; i < 5; i++)
    printf ("\n %d %d ---element", list[i], i);
}
```

The output of this program would be:

This list is...

```
          20          0……element
          30          1……element
          35          2……element
          36          3……element
          39          4……element
```

The above program can also be written by pointer method as shown below:

```
/* Pointer method of processing an array */
#include <stdio.h>
main()
{
  static int list [] = {20, 30, 35, 36, 39};
  int i;
  printf ("\n The list is …");
```

```
  for (i = 0; i < 5; i++)
    printf ("\n %d %d ---element", *(list +i), i);
}
```

It may be noted in the above program that we have used the term `* (list + i)` to print an *i*th element of the array. Let us analyse this term. Since list designates the address of zeroth element of the array, we can access its value through value at address operator, i.e., '*'. The following terms are equivalent:

```
-> *list ≡ *(list + 0) ≡ list [0]
-> *(list + 1) ≡ list [1] and so on
```

Thus, we can refer to *i*th element of array list by either of the following ways:

```
*(list +i) or *(i + list) or list [i]
```

So far, we have used the name of an array to get its base address and manipulate it. However, there is a small difference between an ordinary pointer and the name of an array. The difference is that the array name is a pointer constant and its contents cannot be changed. Thus, the following operations on list are illegal because they try to change the contents of a list, a pointer constant.

```
    List = NULL;          /* Not allowed */
    List = & Val;         /* Not allowed */
    list++                /* Not allowed */
    list--                /* Not allowed */
```

Consider the following program:

```
#include <stdio.h>
main()
```

```
{
  char text[6] =“Helpme”;
  char *p =“Helpme”;
  printf (“\n %c”, text[3]);
  printf (“\n %c”, p[3]);
}
```

The output of above program would be:

```
p
p
```

From the above program, it looks as if the following declarations are equivalent as `text[3]` and `p[3]` behave in identical manner.

```
char text [6];
char *p;
```

It may be noted that the above declarations are not at all equivalent though the behaviour may be same. In fact, the array declaration ‘`text` [6]’ asks from the complier for six locations of char type whereas the pointer declaration char `*p` asks for a pointer that can point to any variable of following type:

1. char – a character
2. string – a string of a characters
3. Null – nowhere

Consider the following declarations:

```
int list = {20, 30, 35, 36, 39};
int *p;  /* pointer variable p */
p = list;  /* assign the starting address of array list to pointer p */
```

Since p is a pointer variable and has been assigned the starting address of array list, the following operations become perfectly valid on this pointer:

```
p++, p-- etc.
```

Let us now write a third version of the program that prints out the array. We will use pointer variable p in this program.

```
/* Pointer variable method of processing an array */

#include <stdio.h>

main()

{

  static int list [] = {20, 30, 35, 36, 39};

  int *p;

  int i = 0;

  p = list;  /* Assign the starting address of the list
*/

  printf ("\n The list is …");

    while (i < 5)

    {

      printf ("\n %d %d ---element", *p, i);

      i++;

      p++;  /* increment pointer */

    }

}
```

The output of the above program would be as shown below:

```
20          0.....element
30          1.....element
35          2.....element
36          3.....element
39          4.....element
```

From our discussion on arrays, we know that the strings are stored and manipulated as array of characters with last character being a null character (i.e., `'\0'`).

For example, the string ENGINEERING can be stored in an array (say text) as shown in Figure 5.14.

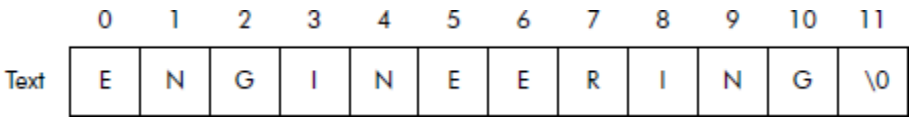


Fig. 5.14 The array called 'text'

We can declare this string as a normal array by the following array declaration:

```
char text [11];
```

Consider the following declaration:

```
char *p;
p = text;
```

In the above set of statements, we have declared a pointer `p` that can point to a character or string of characters. The next statement assigns the starting address of character string '`text`' to the variable pointer `p` (see Figure 5.15).

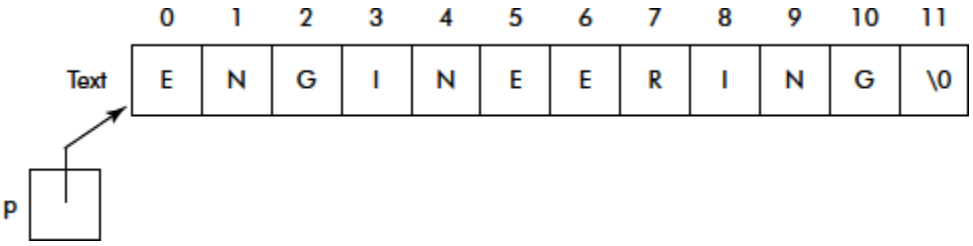


Fig. 5.15 Pointer p points to the array called 'text'

Since `text` is a pointer constant, its contents cannot be changed. On the contrary, `p` is a pointer variable and can be manipulated like any other pointer as shown in the program given below:

```
/* This program illustrates the usage of pointer to a
string */

#include <stdio.h>

main()

{

  char text[] = "ENGINEERING";  /* The string */

  char *p;    /* The pointer */

  p = text;  /* Assign the starting address of string to
p */

  printf ("\n The string..");  /* Print the string */

  while (*p != '\0')

  {printf ("%c", *p);

    p++;

  }
```

```
}
```

Example 2: What would be the output of the following code?

```
#include <stdio.h>
{
  char *ptr;
  ptr = “Nice”;
  printf (“\n% c”, *&ptr [1]);
}
```

Solution: The output would be: i.

Example 3: What would be the output of the following code?

```
#include <stdio.h>
main ()
{
  int list [5] ={2, 5, 6, 0, 9};
  3[list] = 4[list] + 6;
  printf (“%d”, *(list + 3));
}
```

Solution: The output would be 15.

5.4 ARRAY OF POINTERS

Like any other array, we can also have an array of pointers. Each element of such an array can point to a data item such as variable, array etc. For example, consider the declaration given below:

```
float *x [20];
```

This declaration means that `x` is an array of 20 elements and each element is a pointer to a variable of type float. Let us now construct an array of pointers to strings.

```
char *item [ ]{ Chair,
  Table,
  Stool,
  Desk,
};
```

The above declaration gives an array of pointers called item. Each element of item points to a string as shown in Figure 5.16.

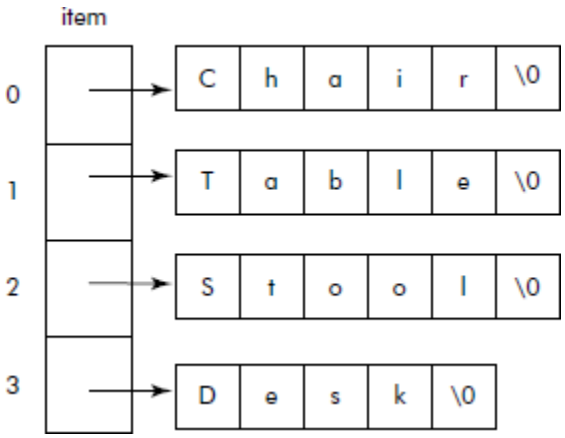


Fig. 5.16 Array of pointers to strings

The advantage of having an array of pointer to strings is that manipulation of strings becomes convenient. For example, the string containing table can be copied into a pointer `ptr` without using `strcpy()` function by the following set of statements:

```
char *ptr; // declare a pointer to a string
```

```
 ptr = item [1]; // assign the appropriate pointer to
ptr
```

It may be noted that once the above set of statements is executed, both the pointers `item[1]` and `ptr`will point to the same string i.e., 'table' as shown in Figure 5.17.

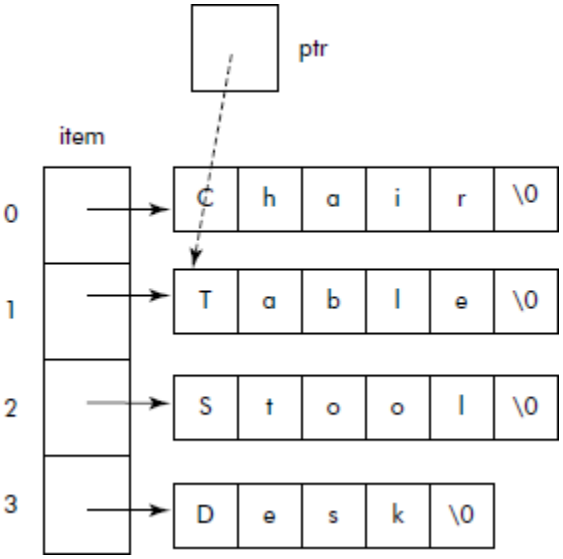


Fig. 5.17 Effect of statement ptr = item[1]

However, the changes made by the pointer `ptr` to the string table will definitely disturb the contents of the string pointed by the pointer `item [1]` and vice versa. Therefore, utmost care should be taken by the programmer while manipulating pointers pointing to the same memory locations.

5.5 POINTERS AND STRUCTURES

We can have a pointer to a structure, i.e., we can also point a pointer to a structure. A pointer to a structure variable can be declared by prefixing the variable name by a '*'. Consider the following declaration:

```
struct xyz {/* structure declaration */

int a;

float y;
```

```
};

  struct xyz abc; /* declare a variable of type xyz */

  struct xyz *ptr; /* declare a pointer ptr to a
variable of type xyz */
```

Once the above declarations are made, we can assign the address of the structure variable `abc` to the structure pointer `ptr` by the following statement:

```
ptr = &abc;
```

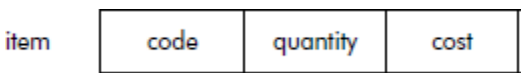
Since `ptr` is a pointer to the structure variable `abc`, the members of the structure can also be accessed through a special operator called `arrow operator`, i.e., '→' (minus sign followed by greater than sign). For example, the members `a` and `y` of the structure variable `abc` pointed by `ptr` can be assigned values 30 and 50.9, respectively by the following statement:

```
ptr -> a = 30;

ptr -> y = 50.9;
```

The `arrow operator '->'` is also known as a `pointer-to-member operator.`

Example 4: Write a program that defines a structure called `item` for the record structure given below. It reads the data of the record through dot operator and prints the record by `arrow operator.` Use suitable data types for the members of the structure called '`item`'.



Solution: We will use a pointer ptr to point to the structure called 'item'. The required program is given below:

```
/* This program demonstrates the usage of an arrow
operator */

#include <stdio.h>

main()

{

  struct item {

    char code[5];

    int Qty;

    float cost;

};

  struct item item_rec;  /* Define a variable of struct
type */

  struct item *ptr;  /* Define a pointer of type struct
*/

      /* Read data through dot operator */

  printf ("\n Enter the data for an item");

  printf ("\nCode:"); scanf ("%s", &item_rec.code);

  printf ("\nQty:"); scanf ("%d", &item_rec.Qty);

  printf ("\nCost:"); scanf ("%f", &item_rec.cost);

      /* Assign the address of item_rec */

  ptr = &item_rec;

      /* Print data through arrow operator */
```

```
  printf ("\n The data for the item…");
  printf ("\nCode : %s", ptr -> code);
  printf ("\nQty : %d", ptr -> Qty);
  printf ("\nCost : %5.2f", ptr -> cost);
}
```

From the above program, we can see that the members of a static structure can be accessed by both dot and arrow operators. However, dot operator is used for simple variable whereas the arrow operator is used for pointer variables.

Example 5: What is the output of the following program?

```
#include <stdio.h>
main()
{
  struct point
    {
      int x, y;
    } polygon[]= {{1,2},{1,4},{2,4},{2,2}};
  struct point *ptr;
  ptr = polygon;
  ptr++;
  ptr -> x++;
  printf ("\n %d", ptr -> x);
}
```

Solution: From the above program, it can be observed that '`ptr`' is a pointer to an array of structures called polygon. The statement `ptr++` moves the pointer from zeroth location (i.e., {1, 2}) to first location (i.e., {1, 4}). Now, the statement `x++` has incremented the field `x` of the structure by one (i.e., 1 has incremented to 2).

The output would be 2.

5.6 DYNAMIC ALLOCATION

We know that in a program when a variable `x` is declared, a storage location is ade available to the program as shown in Figure 5.18.

This memory location remains available, even if it is not needed, to the program as long as the program is being executed. Therefore, the variable `x` is known as a **static variable**. Dynamic variables, on the other hand, can be allocated from or returned to the system according to the needs of a running program. However, a dynamic variable does not have a name and can be referenced only through a pointer.

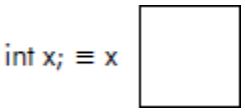


Fig. 5.18 Memory allocation to variable x

Consider the declaration given below:

```
int*p;
```

The above declaration gives us a dangling pointer p that points to nowhere (see Figure 5.19). Now, we can connect this dangling pointer either to a static variable or a dynamic variable. In our previous discussion on pointers, we have always used pointers with static variables. Let us now see how the pointer can be connected to a dynamic variable.

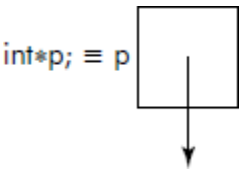


Fig. 5.19 The dangling pointer p

In ‘C’, dynamic memory can be allocated by a function called `malloc()`. The format of this function is given below:

```
(<type> *) (malloc (<size>))
```

where	`<type:>`	is the type of variable, which is required at run time.
	`malloc:`	is a reserved word indicating that memory allocation is to be done.
	`<size>:`	is the size of memory to be allocated. In fact, the size of any variable can be obtained by `sizeof()` function.

Consider the following declaration:

```
int *p; int Sz;

Sz = sizeof (int);

P = (int*) malloc (Sz);
```

The above set of statements is executed in the following manner;

1. The pointer `p` is created. It is dangling, i.e., it is not connected anywhere (see Figure 5.20 (a))
2. A variable called `Sz` is declared to store the size of dynamic variable.
3. The size of dynamic variable (int in this case) is stored in the variable called `Sz`.
4. A dynamic variable of type integer is taken through `malloc()` function and the dangling pointer`p` is connected to the variable (see Figure 5.20 (b)).

It may be noted from Figure 5.20 (b) that a dynamic variable is anonymous in the sense that it has no name and, therefore, it can be referenced only through the

pointer `p`. For example, a value (say 50) can be assigned to the dynamic variable by the following statement:

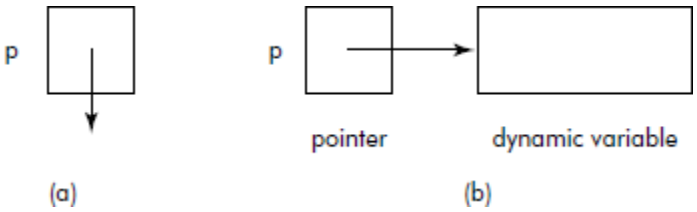


Fig. 5.20 The effect of malloc() function

```
*p = 50;
```

The above statement means that it is desired to store a value 50 in a memory location pointed to by the pointer `p`. The effect of this statement is shown in Figure 5.21.

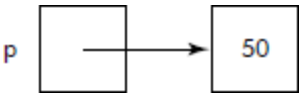


Fig. 5.21 The effect of assignment of *p = 50

It may be observed that p is simply a pointer or an address and `*p` gives the access to the memory location or the variable where it is pointing to. It is very important to learn the difference between the pointer and its associated dynamic variable. Let us consider the following program segment:

```
int *Aptr, *Bptr;  /* declare pointers */

Aptr = (int *) malloc (sizeof int));  /* allocate
dynamic memory */

Bptr = (int *) malloc (sizeof (int));

*Aptr = 15;

*Bptr = 70;
```

The outcome of the above program segment would be as shown in Figure 5.22.

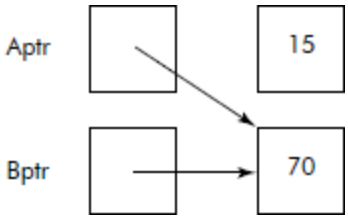


Fig. 5.22 Value assignment through pointers

Let us now see the effect of the following assignment statement on the memory allocated to the pointers `Aptr and Bptr.`

```
Aptr = Bptr;
```

The above assignment makes `Aptr` point to the same location or variable which is already being pointed by `Bptr` as shown in Figure 5.23. Thus, both `Aptr` and `Bptr` are now pointing to the same memory location.

A close observation would reveal that the dynamic variable containing value 15 is now lost in the sense that it is neither available to `Aptr` any more and nor it is available to the system. The reason is that the system gave it to the pointer `Aptr` and now `Aptr` is pointing to some other location. This dynamic variable, being anonymous, (i.e., without name) is no more accessible. Even if we want to reverse the process, we cannot simply do it. This phenomenon of loosing dynamic variables is known as **memory bleeding**. Thus, utmost care should be taken while manipulating the dynamic variables. It is better to return a dynamic variable whenever it is not required in the program.

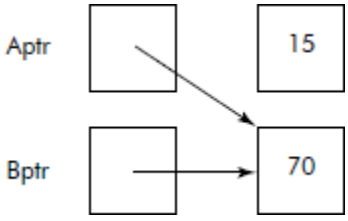


Fig. 5.23 Both pointers pointing to the same location

A dynamic variable can be returned to the system by function called `free().` The general form of usage of this operator is given below:

```
free (p_var);
```

where `free:` is a reserved word

`p_var`: is the pointer to the dynamic variable to be returned to the system.

Let us consider pointers shown in Figure 5.23. If it is desired to return the dynamic variable pointed by pointer `Bptr,` we can do so by the following statement:

```
free (Bptr);
```

Once the above statement is executed, the dynamic variable is returned back to the system and we are left with two dangling pointers `Bptr` and `Aptr` as shown in Figure 5.24. The reason for this is obvious because both were pointing to the same location. However, the dynamic variable containing value 15 is anonymous and not available anymore and, hence, a total waste of memory.

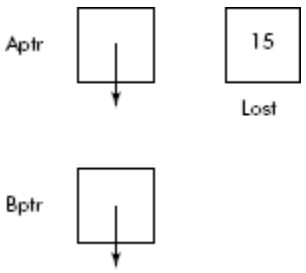


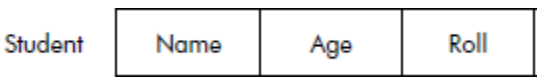
Fig. 5.24 The two dangling pointers and a lost dynamic memory variable

Example 6: Write a program that dynamically allocates an integer. It initializes the integer with a value, increments it, and print the incremented value.

Solution: The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
main()
{
  int *p;  /* A pointer to an int */
  int Sz;
    /* Compute the size of the int */
  Sz = sizeof (int);
    /* Allocate a dynamic int pointed by p */
  p = (int *) malloc (Sz);
  printf ("\n Enter a value :");
  scanf ("%d", p);
  *p = *p + 1;
    /* Print the incremented value */
  printf ("\n The Value = %d", *p);
  free(p);
}
```

Example 7: Write a program that dynamically allocates a structure whose structure diagram is given below. It reads the various members of the structure and prints them.



Solution: We will use the following steps to write the required program:

1. Declare a structure called student.
2. Declare a pointer ptr to the structure student.
3. Compute the size of the structure.
4. Ask for dynamic memory of type student pointed by the pointer ptr.
5. Read the data of the various elements using arrow operator.
6. Print the data.
7. Return the dynamic memory pointed by ptr.

The required program is given below:

```
#include <stdio.h>

#include <alloc.h>

main()

{

  struct student {  /* Declare the structure */

  char name[15];

  int age;

  int roll;

  };

  struct student *ptr;

  int Sz;

    /* Compute the size of struct */

  Sz = sizeof(struct student);

    /* Ask for dynamic memory of type student*/

  ptr = (struct student *) malloc(Sz);

  printf ("\n Enter the data of the student");
```

```
  printf ("\nName:"); /*fflush(stdin);*/ gets(ptr ->
name);

  printf ("\nAge:"); scanf ("%d", &ptr -> age);

  printf ("\nRoll:"); scanf ("%d", &ptr -> roll);

  printf ("\n The data is...");

  printf ("\nName :"); puts(ptr -> name);

  printf ("\nAge :%d", ptr -> age);

  printf ("\nRoll :%d", ptr -> roll);

  free (ptr);

}
```

Note: An array can also be dynamically allocated as demonstrated in the following example.

Example 8: Write a program that dynamically allocates an array of integers. A list of integers is read from the keyboard and stored in the array. The program determines the smallest in the list and prints its location in the list.

Solution: The solution to this problem is trivial and the required program is given below:

{coderipe}

```
/* This program illustrates the usage of dynamically
allocated array */

#include <stdio.h>

main()

{

  int i, min, pos;
```

```
  int *list;
  int Sz, N;
  printf ("\n Enter the size of the list:");
  scanf ("%d", &N);
  Sz = sizeof(int) *N ;  /* Compute the size of the list
*/
  /* Allocate dynamic array size N */
  list = (int*) malloc (Sz);
  printf ("\n Enter the list");
      /* Read the list */
  for (i = 0; i < N; i++)
    {
      printf ("\n Enter Number:");
      scanf ("%d", (list + i));
    }
  pos = 0;  /* Assume that the zeroth element is min */
  min = *(list + 0);
      /* Find the minimum */
  for (i = 1; i < N; i++)
    {
      if (min > *(list + i))
        {min = *(list + i);
```

```
        pos = i;

        }

    } /* Print the minimum and its location */

  printf ("\n The minimum is = %d at position = %d",
min, pos);

  free (list);

}
```

From the above example, it can be observed that the size of a dynamically allocated array can be specified even at the run time and the required amount of memory is allocated. This is in sharp contrast to static arrays for whom the size have to be declared at the compile time.

5.6.1 Self Referential Structures

When a member of a structure is declared as a pointer to the structure itself then the structure is called a self referential structure. Consider the following declaration:

```
struct chain {

  int val;

  struct chain *p;

};
```

The structure '*chain*' consists of two members: `val` and `p`. The member val is a variable of type `int`whereas the member `p` is a pointer to a structure of type *chain*. Thus, the structure *chain* has a member that can point to a structure of type chain or may be itself. This type of self referencing structure can be viewed as shown in Figure 5.25.

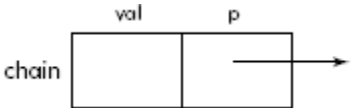


Fig. 5.25 Self referential structure chain

Since pointer p can point to a structure variable of type chain, we can connect two such structure variables A and B to obtain a linked structure as shown in Figure 5.26.

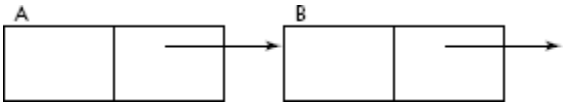


Fig. 5.26 Linked structure

The linked structure given in Figure 5.26 can be obtained by the following steps:

1. Declare structure chain.
2. Declare variable A and B of type chain.
3. Assign the address of structure B to member p of structure A.

These above steps have been coded in the program segment given below:

```
struct chain {  /* Declare structure chain */

  int val;

  chain *p;

};

struct chain A, B; /* Declare structure variables A and
B */

A.p = &B;  /* Connect A to B */
```

From Figure 5.26 and the above program segment, we observe that the pointer p of structure variable Bis dangling, i.e., it is pointing to nowhere. Such pointer can be assigned to NULL, a constant indicating that there is no valid address in this pointer. The following statement will do the desired operation:

```
  B.p = NULL;
```

The data elements in this linked structure can be assigned by the following statements:

```
A.val = 50;

B.val = 60;
```

The linked structure now looks like as shown in Figure 5.27.

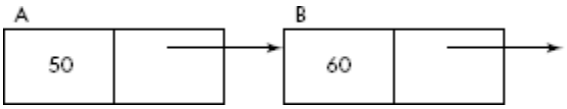


Fig. 5.27 Value assignment to data elements

We can see that the members of structure `B` can be reached by two methods:

1. From its variable name `B` through dot operator.
2. From the pointer `p` of variable `A` because it is also pointing to the structure `B`. However, in this case the arrow operator is needed.

Consider the statements given below:

```
printf ("\n the contents of member val of B = %d",
B.val);

printf ("\n the contents of member val of B = %d", A.p
-> val);
```

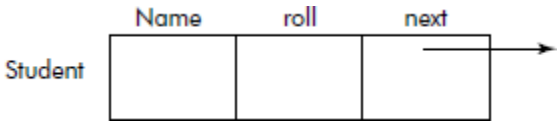
Once the above statements are executed, the output would be:

```
The contents of member val of B = 60

The contents of member val of B = 60
```

The linked structures have great potential and can be used in numerous programming situations such as lists, trees, etc.

Example 9: Write a program that uses self referential structures to create a linked list of nodes of following structure. While creating the linked list, the student data is read. The list is also travelled to print the data.



Solution: We will use the following self referential structure for the purpose of creating a node of the linked list.

```
struct stud{

  char name [15];

  int roll;

  struct stud next;

};
```

The required linked list would be created by using three pointers: `first, far and back.` The following algorithm would be employed to create the list:

Step

1. Take a new node in pointer called first.
2. Read first name and first roll.
3. Point back pointer to the same node being pointed by first, i.e., back 5 first.
4. Bring a new node in the pointer called far.
5. Read far name and far roll.
6. Connect next of back to for, i.e., back next 5 far.
7. Take back to far, i.e., back 5 far.
8. Repeat steps 4 to 7 till whole of the list is constructed.
9. Point next of far to `NULL,` i.e., far next = `NULL.`
10. Stop.

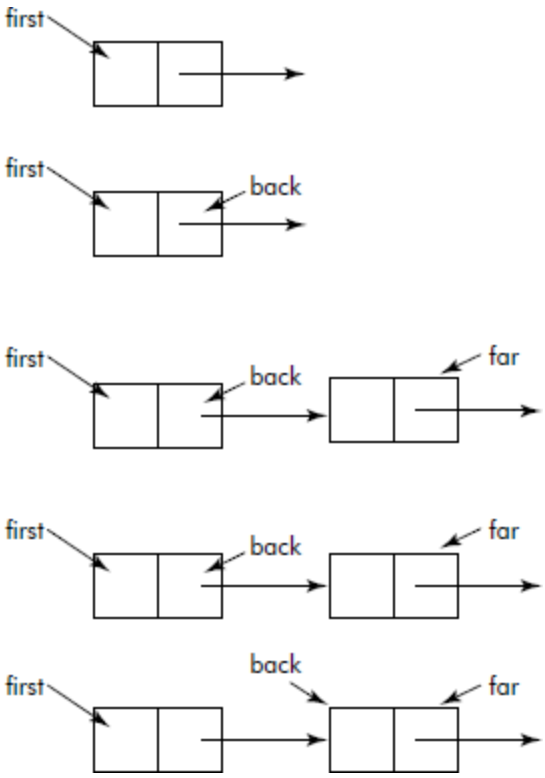


Fig. 5.28 Creation of a linked list

The required program is given below:

```
#include <stdio.h>
#include <alloc.h>
main()
{
  struct student {
     char name [15];
     int roll;
     struct student *next;
     };
  struct student *First, *Far, *back;
```

```
  int N, i;

  int Sz;

  Sz = sizeof (struct student);

  printf ("\n Enter the number of students in the
class");

  scanf ("%d", &N);

      /* Take first node */

  First = (struct student *) malloc(Sz);

      /* Read the data of the first student */

  printf ("\n Enter the data");

  printf ("\n Name : "); fflush(stdin); gets(First ->
name);

  printf ("\n Roll : "); scanf("%d", &First -> roll);

      /* point back where First points */

  back = First;

  for (i =2; i <=N; i++)

  {    /* Bring a new node in Far */

    Far = (struct student *) malloc (Sz);

    /* Read Data of next student */

    printf ("\n Name: "); fflush(stdin); gets(Far ->
name);

    printf ("\n Roll: "); scanf("%d", &Far -> roll);

      /* Connect Back to Far */
```

```
    back -> next = Far;
    /* point back where Far points */
    back = Far;
  }  /* Repeat the process */
  Far -> next = NULL;
      /* Print the created linked list */
  Far = First;  /* Point to the first node */
  printf ("\n The Data is …");
  while (Far != NULL)
  {
    printf ("\n Name: %s , Roll : %d", Far -> name, Far
-> roll);
      /* Point to the next node */
    Far = Far -> next;
  }
}
```

Example 10: Modify the program developed in Example 1 such that

1. assume that the number of students in the list is not known.
2. the list of students is split into the two sublists pointed by two pointers: `front_half` and`back_half`. The `front_half` points to the front half of the list and the `back_half` to the back half. If the number of students is odd, the extra student should go in the front list.

Solution: The list of students would be created in the same fashion as done in `Example 1`. The following steps would be followed to split the list in the desired manner:

1. Travel the list to count the number of students.
2. Compute the `middle` of the list.
3. Point the pointer called `front_half` to first and travel the list again starting from the first and reach to the `middle.`
4. Point a pointer called `back-half` to the node that succeeds the middle of the list.
5. Attach `NULL` to the next pointer of the middle node.
6. Print the two lists, i.e., pointed by `front-half` and `back-half.`

The required program is given below:

```
#include <stdio.h>

#include <alloc.h>

main()

{

  struct student {

    char name [15];

    int roll;

    struct student *next;

  };

  struct student *First, *Far, *back;

  struct student *front_half, *back_half;

  int N, i, count, middle;

  int Sz;

  Sz = sizeof (struct student);

  printf ("\n Enter the number of students in the
class");
```

```
  scanf (“%d”, &N);
    /* Take first node */
  First = (struct student *) malloc (Sz);
    /* Read the data of the first student */
  printf (“\n Enter the data”);
  printf (“\n Name: “); fflush(stdin); gets(First ->
name);
  printf (“\n Roll: “); scanf(“%d”, &First -> roll);
  First -> next =NULL;
    /* point back where First points */
  back = First;
  for (i = 2; i <=N; i++)
  {  /* Bring a new node in Far */
  Far = (struct student *) malloc (Sz);
    /* Read data of next student */
  printf (“\n Name: ”); fflush(stdin); gets(Far ->
name);
  printf (“\n Roll: ”); scanf(“%d”, &Far -> roll);
    /* Connect Back to Far */
  back -> next = Far;
    /* Point back where Far points */
  back = Far;
  }  /* Repeat the process */
```

```
Far -> next = NULL;
  /* Count the number of nodes */
Far = First; /* Point to the first node */
count = 0;
while (Far != NULL)
{
  count ++;
  Far = Far -> next;
} /* Split the list */
if ( count == 1 )
{
  printf ("\n The list cannot be split");
  front_half = First;
}
  else
{
    /* Compute the middle */
  if ( (count % 2) == 0) middle = count/2;
  else middle = count/2 + 1;
    /* Travel the list and split it */
  front_half = First;
  Far = First;
```

```
    for (i =1; i < = middle; i++)
    {
      back = Far;
      Far = Far -> next;
    }
  back_half = Far;
  back -> next = NULL;
  }
      /* Print the two lists */
  Far = front_half;
  printf ("\n The Front Half…:");
  for (i = 1; i < = 2; i++)
  {
    while (Far != NULL)
  {
    printf("Name: %s, Roll: %d ", Far -> name, Far ->
roll);
    Far = Far -> next;
  }
  if (count == 1 || i == 2) break;
  printf ("\n The Back Half…: ");
  Far = back_half;
```

```
  }
}
```

FREQUENTLY ASKED QUESTIONS

1. What is meant by a pointer?
   Ans. It is a variable which can only store the address of another variable.
2. Are the expressions `(*p)++ and ++*p` equivalent?
   Ans. Yes
3. What happens if a programmer assigns a new address to a dynamic variable before he either assigns its address to another pointer or deletes the variable?
   Ans. The dynamic variable will be lost, i.e., it becomes inaccessible to program as well as the operating system.
4. What is meant by memory bleeding?
   Ans. The lost dynamic variables lead to memory bleeding.
5. Pointers always contain integers. Comment.
   Ans. We know that a pointer contains an address of other variables. Since an address can only be a whole number or integer, the pointers contain integers only.
6. What does the term `**j` mean?
   Ans. The term `**j` means that `j` is a pointer to a pointer.
7. Are the expressions `*p++ and ++*p` equivalent?
   Ans. No. The expression `*p++` returns the contents of the variable pointed by `p` and then increments the pointer. On the other hand, the expression `++*p` returns the incremented contents of the variable pointed by `p`.
8. Differentiate between an uninitialized pointer and a `NULL` pointer.

   Ans. An un initialized pointer is a dangling pointer which may contain any erroneous address. A `NULL` pointer does not point to any address location.

TEST YOUR SKILLS

1. Give the output of the following program:

```
#include <stdio.h>
main()
{
  int i, j;
  char *str = "CALIFORNIA";
  for (i = 0; str[i]; i++)
```

```
  {
    for (j = 0; j <= i; j++)
    printf (“%c”, str[j]);
    printf (“\n”);
  }
}
```

Ans. The output would be

```
C
CA
CAL
CALI
CALIF
CALIFO
CALIFOR
CALIFORN
CALIFORNI
CALIFORNIA
```

2. What would be the output of the following?

```
#include <stdio.h>
main()
{
  int a;
  *& a = 50;
  printf (“%d”, a);
}
```

Ans. The output would be `50`.

3. What would be the output of the following code?

```
#include <stdio.h>
main()
{int i = 50;
int *j = &i;
printf (“\n %d”, ++ *(j));
}
```

Ans. The output would be `51`.

4. What would be the output of the following?

```
#include <stdio.h>
main()
{
  char *ptr = "abcd";
  char ch = ++ *ptr ++;
  printf ("\n %c", ch);
}
```

Ans. The output would be `b`.

5. What would be the output of the following code? Assume that the array starts at location 5714 in the memory.

```
#include <stdio.h>
main()
{
  int tab [3][4] = {5, 6, 7, 8,
    1, 2, 3, 4,
    9, 10, 0, 11};
  printf ("\n %d %d", *tab[0] + 1, *(tab[0] + 1));
  printf ("\n %d", *(* (tab + 0) + 1));
}
```

Ans. The output would be

```
6 6
6
```

6. What would be the output of the following code?

```
#include <stdio.h>
main()
{
  struct val {
    int Net;
  };
  struct val x;
  struct val *p;
```

```
  p = &x;
  p -> Net = 50;
  printf ("\n %d", ++ (x.Net));
}
```

Ans. The output would be `50`.

7. What would be the output of the following code?

```
#include <stdio.h>
main()
{
  struct val {
    int Net;
  };
    struct val x;
    struct val *p;
    p = &x;
    p -> Net = 50;
    printf ("\n %d", (x.Net)++);
}
```

Ans. The output would be `21`.

EXERCISES

1. Explain the '`&`' and '`*`' operators in detail with suitable examples.
2. Find the errors in the following program segments:

1.
```
int val = 10
int * p;
p = val;
```

2.
```
char list [10];
char p;
list = p;
```

3. (i) What will be the output of the following program:

```
#include <stdio.h>
main()
{
```

```
  int val = 10;
  int *p, **k;
  p = &val;
  k = &p;
  printf ("\n %d %d %d %d", p, *p, *k, **k);
}
```

(ii)

```
#include <stdio.h>
main()
{
  char ch;
  char *p;
  ch = 'A';
  p = &ch;
  printf ("\n %c %c", ch, (*p)++);
}
```

4. What will be the output of the following:

```
#include <stdio.h>
main()
{
  int list[5], i;
  *list = 5;
  for (i = 1; i < 5; i++)
    *(list + i) =*(list + i – 1)*i;
  printf ("\n");
  for (i = 0; i < 5; i++)
    printf ("%d ",*(list + i)) ;
}
```

5. Find the errors in the following program segment:

```
struct xyz {
  char *A;
  char *B;
  int val;
};
  xyz s;
```

```
A = "First text";
B = "Second text";
```

6. What will be the output of the following program segment?

```
#include <stdio.h>
main()
{
  struct Myrec {
    char ch;
    struct Myrec *link;
  };
  struct Myrec x, y, z, *p;
  x.ch = 'x';
  y.ch = 'y';
  z.ch = 'z';
  x.link = &z;
  y.link = &x;
  z.link = NULL;
  p = &y;
  while (p!= NULL)
  {
    printf ("%c ", p->ch);
    p = p -> link;
  }
}
```

7. What will be the output of the following program?

```
#include <stdio.h>
main()
{
  float cost [] = {35.2, 37.2, 35.42, 36.3};
  float *ptr[4];
  int i;
  for (i = 0; i < 4; i++)
    *(ptr +i) = cost + i;
  for (i = 0; i < 4; i++)
    printf ("%f ",*(*(ptr +i)));
}
```

Assume the base address of the array cost to be 2025.

8. What will be the output of the following program:

```
#include <stdio.h>
main()
{
  char item[]="COMPUTER";
  int i;
  for (i = 7; i > = 0 ; i--)
  printf ("%c", *(item + i));
}
```

- **Copy**
- **Add Highlight**
- **Add Note**

back to top

- Sign Out