Example 11.1

Write a program to print the address of a variable along with its value.

The program shown in Fig.11.4, declares and initializes four variables and then prints out these values with their respective storage locations. Notice that we have used %u format for printing address values. Memory addresses are unsigned integers.

ACCESSING ADDRESSES OF VARIABLES

Program

```
main()
{
    char   a;
    int    x;
    float  p, q;

    a  = 'A';
    x  = 125;
    p  = 10.25, q = 18.76;

    printf("%c is stored at addr %u.\n", a, &a);
    printf("%d is stored at addr %u.\n", x, &x);
    printf("%f is stored at addr %u.\n", p, &p);
    printf("%f is stored at addr %u.\n", q, &q);

}
```

Output

```
A is stored at addr 4436.
125 is stored at addr 4434.
10.250000 is stored at addr 4442.
18.760000 is stored at addr 4438.
```

Fig.11.4 *Accessing the address of a variable*

Example 11.2

Write a program to illustrate the use of indirection operator '*' to access the value pointed to by a printer.

The program and output are shown in Fig.11.5. The program clearly shows how we can access the value of a variable using a pointer. You may notice that the value of the pointer **ptr** is 4104 and the value it points to is 10. Further, you may also note the following equivalences:

```
x = *(&x) = *ptr = y
&x = &*ptr
```

ACCESSING VARIABLES USING POINTERS

Program

```
main()
{
    int   x, y;
    int   *ptr;
    x = 10;
    ptr = &x;
    y = *ptr;

    printf("Value of x is %d\n\n",x);
    printf("%d is stored at addr %u\n", x, &x);
    printf("%d is stored at addr %u\n", *&x, &x);
    printf("%d is stored at addr %u\n", *ptr, ptr);
    printf("%d is stored at addr %u\n", y, &*ptr);
    printf("%d is stored at addr %u\n", ptr, &ptr);
    printf("%d is stored at addr %u\n", y, &y);

    *ptr = 25;
    printf("\nNow x = %d\n",x);

}
```

Output

```
Value of x is 10

10   is stored at addr 4104
10   is stored at addr 4104
10   is stored at addr 4104
10   is stored at addr 4104
4104 is stored at addr 4106
10   is stored at addr 4108

Now x = 25
```

Fig.11.5 *Accessing a variable through its pointer*

Example 11.3

Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig.11.7 shows how the pointer variables can be directly used in expressions. It also illustrates the order of evaluation of expressions. For example, the expression

4* - *p2 / *p1 + 10

is evaluated as follows:

((4 * (-(*p2))) / (*p1)) + 10

When *p1 = 12 and *p2 = 4, this expression evaluates to 9. Remember, since all the variables are of type int, the entire evaluation is carried out using the integer arithmetic.

ILLUSTRATION OF POINTER EXPRESSIONS

Program

```
main()
{
    int  a, b, *p1, *p2, x, y, z;

    a  = 12;
    b  =  4;
    p1 = &a;
    p2 = &b;

    x  =  *p1 * *p2 - 6;
    y  =  4*  - *p2 / *p1 + 10;

    printf("Address of a = %u\n", p1);
    printf("Address of b = %u\n", p2);
    printf("\n");
    printf("a = %d, b = %d\n", a, b);
    printf("x = %d, y = %d\n", x, y);

    *p2  = *p2 + 3;
    *p1  = *p2 - 5;
    z    = *p1 * *p2 - 6;

    printf("\na = %d, b = %d,", a, b);
    printf(" z = %d\n", z);
}
```

Output

```
Address of a = 4020
Address of b = 4016

a = 12, b = 4
x = 42, y = 9
a = 2, b = 7, z = 8
```

Fig.11.7 *Evaluation of pointer expressions*

Example 11.4

Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in Fig.11.8 illustrates how a pointer can be used to traverse an array element. Since incrementing an array pointer causes it to point to the next element, we need only to add one to **p** each time we go through the loop.

POINTERS IN ONE-DIMENSIONAL ARRAY

Program

```
main()
{
    int *p, sum, i;
    int x[5] = {5,9,6,3,7};

    i  = 0;
    p  = x;          /* initializing with base address of x */

    printf("Element   Value   Address\n\n");
    while(i < 5)
    {
       printf(" x[%d] %d %u\n", i, *p, p);
       sum = sum + *p;   /* accessing array element  */
       i++, p++;        /* incrementing pointer     */
    }
    printf("\n  Sum    =  %d\n", sum);
    printf("\n  &x[0]  =  %u\n", &x[0]);
    printf("\n  p      =  %u\n", p);
}
```

Output

```
Element    Value     Address
  x[0]       5         166
  x[1]       9         168
  x[2]       6         170
  x[3]       3         172
  x[4]       7         174

Sum    =  55
&x[0]  =  166
p      =  176
```

Fig.11.8 *Accessing array elements using the pointer*

Example 11.5

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.**

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

POINTERS AND CHARACTER STRINGS

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*

Example 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*

Example 11.7

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.11.12. 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.

ILLUSTRATION OF POINTERS TO FUNCTIONS

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.11.12 *Use of pointers to functions*

Example 11.8

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.11.13. 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.

POINTERS TO STRUCTURE VARIABLES

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.11.13 *Pointer to structure variables*