CHARUTAR VIDYA MANDAL'S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: FYBCA
Subject: US02CBCA01
(Advanced C Programming and Introduction to Data Structures)

*UNIT – 1 (Usage of Pointers)

**POINTERS

*INTRODUCTION

A pointer is a derived data type in C. It is built from the one of the fundamental data types available in the C. Pointer contain memory address as their values. Memory addresses are the locations in computer memory where program instructions and data are stored. So, we can use pointers to access and manipulate data stored in memory.

BENEFITS OF POINTERS

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function through function arguments.
3. Pointers allow references to functions. So we can pass function as argument to other function.
4. We can save data storage space in memory by using pointer arrays to character strings.
5. Pointers allow C to support dynamic memory management.
6. Pointers is an efficient tool to manipulate dynamic data structure such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. Pointers increase execution speed and reduce execution time.

UNDERSTANDING POINTERS

The computer's memory is a collection of thousands of sequential 'storage cells' as shown in figure 1. Each cell (known as a byte) is identified by a unique address.

The memory addresses in a given computer range from 0 to a maximum value that depends on the memory size. A computer system having 64 K (64 * 1024 = 65,536) memory will have its last address as 65,535.

Memory Cell	Address
	0
	1

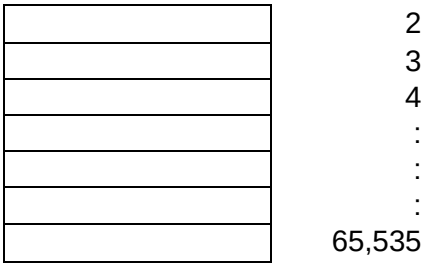


Fig.1 Memory organization

When we declare a variable, the compiler allocates a memory location somewhere in the memory to hold the value of the variable. The compiler associates that address with the variable's name.

When your program uses the variable name, it automatically accesses the proper memory location. The location's address is used, but it is hidden from you, and you need not be concerned with it.

int quantity = 179;

Above statement instructs the system to reserve a location for the integer variable **quantity** and puts the value 179 in that location. Assume that compiler has reserved the address location 5000 for **quantity**.

Note that the address of variable is the address of the first byte occupied by that variable.

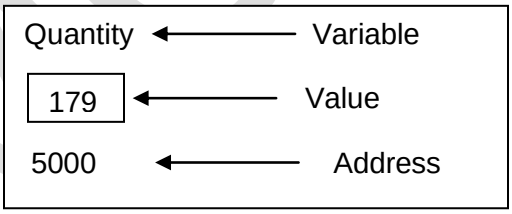


Fig. 2 Representation of a variable

During execution of the program, the system always associates the name **quantity** with 5000. This is something similar to having a house number as well as a house name. We can access to the value 179 by using either the name **quantity** or the address 5000.

CREATING A POINTER

Memory addresses are simply numbers. So they can be assigned to some variables and these variables can be stored in memory, like any other variable. Such variables that hold memory addresses are called **pointer variables**.

Def: A **pointer variable** is nothing but a variable that contains an address, which is a location of another variable in memory.

A pointer is a variable so its value is also stored in the memory in another location.

So, the first step is to declare a variable to hold the address of quantity. Give it the name p. Storage has been allocated for p, but its value is undetermined.

The next step is to store the address of the variable quantity in the variable p. Because p now contains the address of rate, it indicates the location where quantity is stored in memory.

p points to quantity, or is a pointer to quantity.

Suppose, we assign the address of **quantity** to a variable **p**. The link between the variable **p** and **quantity** can be seen in Fig.3. The address of **p** is 5048.

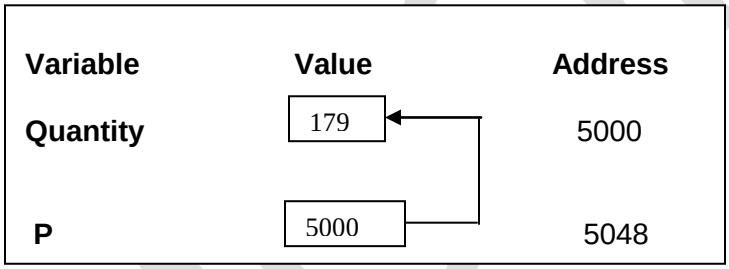
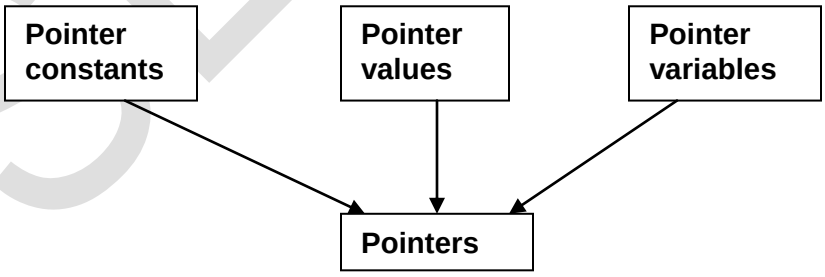


Fig. 3 Pointer variable

Since the value of the variable **p** is the address of the variable **quantity,** we may access the value of **quantity** by using the value of **p** and therefore, we say that the variable **p** 'points' to the variable **quantity.** Thus, **p** gets the name 'pointer'. (Actual values of pointer variables may be different every-time we run the program.)

FUNDAMENTAL CONCEPTS OF POINTERS

Pointers are built on three concepts as illustrated below



Memory addresses within a computer are referred to as **pointer constants**. We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through using the address operator (&).The value obtained is known as a **pointer value**. The pointer value (i.e the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a **pointer variable**.

ACCESING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent. Therefore, address of a Variable is not known to us immediately. This can be determined with the help of the operator **&** available in C.

The operator **&** immediately preceding a variable returns the address of the variable associated with it. For example, the statement

p= &quantity

would assign the address 5000 (the location of quantity) to variable **p**. The **&** operator can be remembered as **'address of'**

The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. &125(pointing at constants).
2. int x[10]
   &x(pointing at array names)
3. &(x+y)(pointing at expressions).

If x is an array, then expressions such as **&x[0] and x[i+3]** are valid and represent the address of 0th and (i+3)rd elements of x.

Example-1 Write a program to print the address of a variable along with its value.

The program shown in figure given below declares an initializes four variables and then prints out these values with their respective storage locations. %u format is used for printing address values. Memory addresses are unsigned integers.

Program

```
main()
{  char a;
   int x;
   float p;
   a =  'A';
   x = 125;
   p = 10.25;
   printf("%c is stored at address %u.\n", a, &a");
   printf("%d is stored at address %u.\n", x, &x);
   printf("%f is stored at address %u.\n", p, &p);
}   // Accessing the address of variable
```

OUTPUT

A is stored at address 4436.
125 is stored at address 4434.
10.250000 is stored at address 4442.

* DECLARING, INITIALIZATION and ACCESSING POINTERS

DECLARING POINTER VARIABLES

In C, every variable must be declared with its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. Pointer variable names follow the same rules as other variables and must be unique.

The declaration of a pointer variable has the following syntax:

data_type *ptrname;

data_type is any of C's variable types and indicates the type of the variable that the pointer points to.

Declaration tells the compiler three things about the variable **ptrname**

1. The asterisk (*) is the indirection operator and tells that the variable name **ptrname** is a pointer name.
2. **ptrname** needs a memory location.
3. **ptrname** points to a variable of type data_type.

Example: **int *p;** /* integer pointer */

Declares the pointer **p** as a pointer variable that points to an integer data type. Remember that the type **int** refers to the data type of the variable being pointed to by **p** and not the type of the value of the pointer. Similarly, the statement

float *x; / * float pointer * /

declares **x** as a pointer to the floating-point variable.

The declarations tell the compiler to allocate memory locations for the pointer variables **p** and **x**. Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:

int *p; **p** | **?** | **----------> ?**

Contains garbage Points to Unknown location

Pointer Declaration Style

Pointer variables are declared to normal variables except for the addition of unary * operator. This symbol can appear anywhere between the type name and printer variable name. Programmers use the following styles:

```
int*    p;    /* style 1 */
int     *p;   /* style 2 */
int  *  p;    /* style 3 */
```

Style2 is becoming more and more popular because it is convenient for multiple declarations in the same statement. Example: int *p, x, *q;

INITIALIZATION OF POINTER VARIABLES

The process of assigning address of a variable to a pointer variable is known as initialization.

Once a pointer variable has been declared, Program must put address in pointer by using the address-of operator, i.e. ampersand (&).

When & is placed before the name of a variable, the address-of operator returns the address of the variable. Therefore, you initialize a pointer with a statement of the form
pointer = &variable;
Example:

```
int   quantity;          /*  declaration of simple int variable */
int  *p;                 /*  declaration  */
p =  &quantity;          /*  initialization  */
```

- We can also combine the initialization with the declaration. That is,

  int *p = &quantity;

  is allowed. The only requirement here is that the variable quantity must be declared before the initialization takes place. Remember, this is initialization of p and not *p.

- It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

  For example,

  int x, *p = &x; /* three in one */

  is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. And also remember that the target variable x is declared first. The statement **int *p = &x, x;** is not valid.

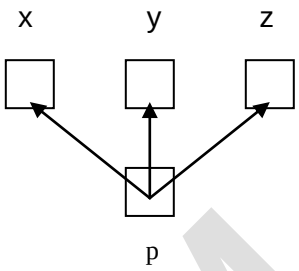
- We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statement is valued:

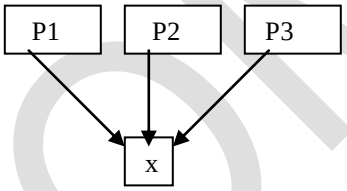
  ```
  int  *p = NULL;
  int  *p = 0;
  ```

  With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

  ```
  int *p = 5360;        / *absolute address */
  ```

- Pointers are flexible.

We can make the same pointer to point to different data variables in different statements.
Example;
int x,y,z, *p;
……..
p = &x;
…….
p =&y;
……
p=&z;
	

We can use different pointers to point to same data variables.
Example;
int x;
int *p1 = &x;
int *p2 = &x;
int *p3 = &x;
	

Note

1. All uninitialized pointers will have some unknown values. They may not be valid addresses or they may point to some values that are wrong. Compilers do not detect these errors. So the programs with uninitialized pointers will produce wrong results. It is therefore important to initialize pointer variables carefully before they are used in the program.
2. Pointer variables always point to the corresponding type of data. When we declare a pointer to be of int type, the system assumes that any address that the pointer will hold will point to an integer variable. Take care to avoid wrong pointer assignments because the compiler will not detect such errors.
   For example,

```
float  a, b;
int x, *p;
p = &a;          /* wrong  */
b = *p;
```

   will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer.

ACCESSING A VARIABLE THROUGH ITS POINTER

After assigning address of a variable to pointer, we can access the value of the variable using the pointer. This is done by using another unary operator ***(asterisk)**, usually known as the **indirection operator or dereferencing operator**.

E.g.

```
int quantity, *p, n;
quantity = 179;
p = &quantity;
n =*p;
```

1. First line declares quantity and n as integer variable and p as a pointer variable pointing to an integer.
2. Second line assigns the value 179 to quantity.
3. Third line assigns the address of quantity to the pointer variable p.
4. Fourth line contains the indirection operator *.

When the operator * is placed before a pointer variable in an expression (on the right- hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address.

In this case, *p returns the value of the variable quantity, because p is the address of quantity. The * can be remembered 'value at address'. Thus the value of n would be 179.

The two statements	p = &quantity; n = *p;
are equivalent to	n = *&quantity;
which is equivalent to	n = quantity;

Example-2 illustrates the distinction between pointer value and the value it points to.

Example-2 Write a program to illustrate the use of indirection operator '*' to access the value pointed to by a pointer.

The program shows that we can access the value of a variable using a pointer. Value of the pointer ptr is 4104 and the value it points to is 10. Note the following equivalences:

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

Program

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

   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",ptr,&ptr);
   printf("%d is stored at addr%u\ny,&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
4104    is stored at addr 4106
10      is stored at addr 4108
```

Accessing a variable through its pointer

The actions performed by the program are illustrated in following figure.

- The statement **ptr=&x** assigns the address of **x** to **ptr** and **y = *ptr** assigns the value pointed to by the pointer **ptr** to **y.**
- ***ptr = 25;** puts the value of 25 at the memory location whose address is the value of **ptr**.
  Value of **ptr** is the address of **x** and therefore the old value of **x** is replaced by 25. This is equivalent to assigning 25 to **x**. This way we can change the value of a variable indirectly using pointer and the **indirection operator.**

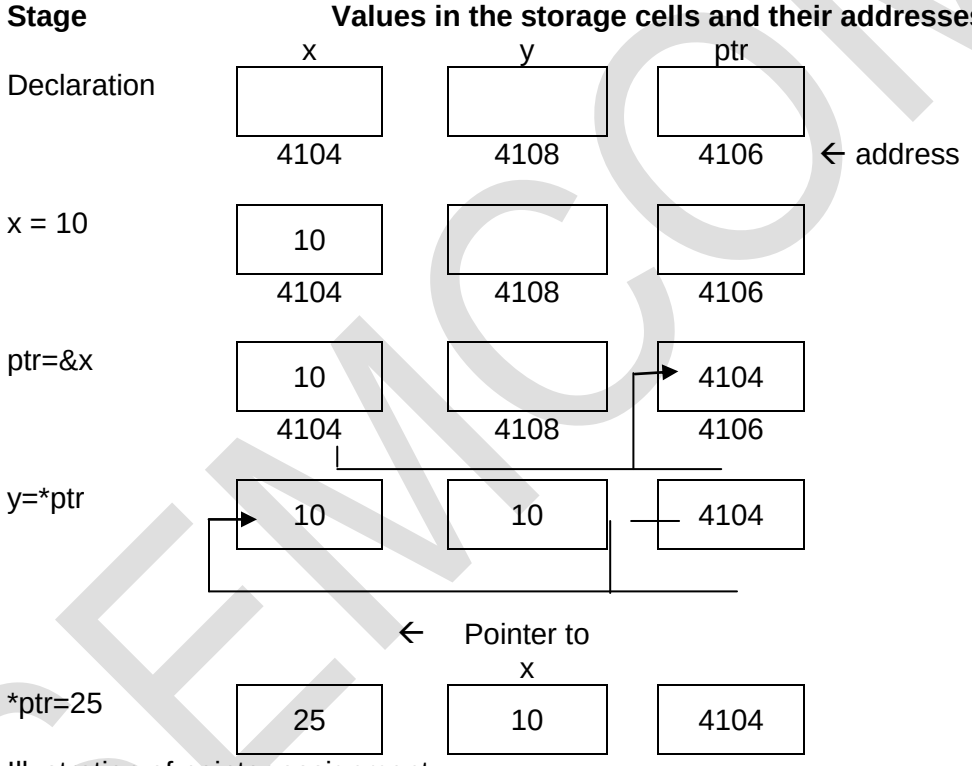


Illustration of pointer assignment

***POINTER ARITHMETIC**

POINTER INCREMENTS AND SCALE FACTOR

Pointers can be incremented like

p1=p2+2;
p1=p1+1; And so on.

p1++; will cause the pointer **p1** to point to the next value of its type. For example, if **p1** is an integer pointer with an initial value, say 2800, then after the operation **p1=p1+1**, the value of **p1** will be 2802, and not 2801.

Means when we increment a pointer, its value is incremented by the 'length' of the data type that it points to. This length called the scale factor.

The number of bytes used to store various data types depends on the system and can be found by making use of the **sizeof** operator. For example, if x is a variable, then **sizeof (x)** returns the number of bytes needed for the variable. (Systems like Pentium use 4 bytes for storing integers and 2 bytes for short integers.)

If you have a pointer to the first array element; the pointer must be incremented by an amount equal to the size of the data type stored in the array. You can access array elements using pointer arithmetic.

INCREMENTING POINTERS

When you increment a pointer, you are increasing its value. For example, when you increment a pointer by 1, pointer arithmetic automatically increases the pointer's value so that it points to the next array element.

C knows the data type that the pointer points to (from the pointer declaration) and increases the address stored in the pointer by the size of the data type.

Suppose that ptr_to_int is a pointer variable to some element of an int array.

If you execute the statement: ptr_to_int++;
the value of ptr_to_int is increased by the size of type int (usually 2 bytes), and ptr_to_int now points to the next array element.

Likewise, if ptr_to_float points to an element of a type float array, the statement
ptr_to_float++;
increases the value of ptr_to_float by the size of type float (usually 4 bytes).

It is true for increments greater than 1. If you add the value n to a pointer, C increments the pointer by n array elements of the associated data type. Therefore,

ptr_to_int += 4;
increases the value stored in ptr_to_int by 8 (assuming that an integer is 2 bytes), so it points four array elements ahead.

Likewise, ptr_to_float += 10;
increases the value stored in ptr_to_float by 40 (assuming that a float is 4 bytes), so it points 10 array elements ahead.

DECREMENTING POINTERS

If you decrement a pointer with the -- or -= operators, pointer arithmetic automatically adjusts for the size of the array elements.

Example 3 shows how pointer arithmetic can be used to access array elements. By incrementing pointers, the program can step through all the elements of the arrays efficiently.

Example-3 Using pointer arithmetic and pointer notation to access array elements.

```
#include <stdio.h>
void main()
{
        int i_array[5] = { 1,2,3,4,5 };
        int *i_ptr, count;
        float f_array[5] = { .1, .2, .3, .4, .5 };
        float *f_ptr;

        f_ptr = f_array;
        i_ptr = i_array;

        for (count = 0; count < ; count++)
            printf("%d  %f \n", *i_ptr++, *f_ptr++);
}
```

Output:

```
1    0.100000
2    0.200000
3    0.300000
4    0.400000
5    0.500000
```

POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if **p1** and **p2** are properly declared and initialized pointers, then the following statements are valid.

```
Y = *p1 * * p2;        same as (*p1) * (*p2)
sum = sum + *p1;
z = 5 * - *p2 / *p1;      same as (5 * (- (*p2) ) ) / (*p1)
*p2 = *p2 + 10;
```

Note that there is a blank space between / and * in the item3 above. The following is wrong.

```
z =5* - *p2 / *p1;
```

The symbol /* is considered as the beginning of a comment and therefore the statement fails.

C allows us to add integers to or subtract integers from pointers, as well as to subtract one pointer from another. p1+4, p2-2 and p1-p2 are all allowed.

If p1 and p2 are both pointers to the same array, the p2-p1 gives the number of elements between **p1** and **p2**.

We may also use short-hand operators with the pointers.

```
p1++;
-p2;
sum+=*p2;
```

Pointers can also be compared using the relational operators. The expressions such as **p1>p2, p1==p2, and p1! =p2** are allowed.

Any comparison of pointers that refers to separate and unrelated variables makes no sense. Comparisons can be used meaningfully in handling arrays and strings.

We may not use pointers in division or multiplication. For example, expressions such as

p1 / p2 or p1 * p2 or p1 / 3

are not allowed. Similarly, two pointers cannot be added. That is, p1+p2 is illegal.

Example-4 Write a program to illustrate the use of pointers in arithmetic operations.

The program in Fig 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.

Program

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

Evaluation of pointer expression

POINTER OPERATIONS

You can do a total of six operations with a pointer.

Operation	Description
Assignment	You can assign a value to a pointer. The value should be an address, obtained with the address-of operator (&) or from a pointer constant (array name).
Indirection	The indirection operator (*) gives the value stored in the pointed-to location.
Address of	You can use the address-of operator to find the address of a pointer, so you can have pointers to pointers.
Incrementing	You can add an integer to a pointer in order to point to a different memory location.
Decrementing	You can subtract an integer from a pointer in order to point to a different memory location.
Differencing	You can subtract an integer from a pointer in order to point to a different memory location.
Comparison	Valid only with two pointers that point to the same array.

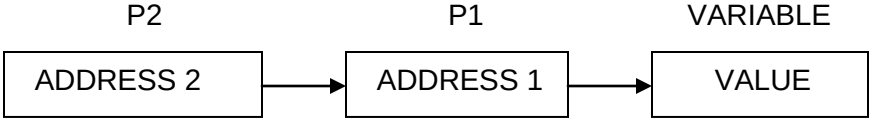
Rules of pointer operations

The following rules apply when performing operations on pointer variables.

1. A pointer variable can be assigned the address of another variable.
2. A pointer variable can be assigned the values of another pointer variable.
3. A pointer variable can be initialized with NULL or zero value.
4. A pointer variable can be pre-fixed or post-fixed with increment or decrement operators.
5. An integer value may be added or subtracted from a pointer variable.
6. When two pointers point to the same array, one pointer variable can be subtracted from another.
7. When two pointers point to the objects of the same data type, they can be compared using relational operators.
8. A pointer variable cannot be multiplied by a constant.
9. Two pointer variables cannot be added.
10. A value cannot be assigned to an random address.(i.e. &x =10; is illegal)

*POINTER TO POINTER (CHAIN OF POINTERS)

It is possible to make a pointer to point to another pointer, thus creating a chain of pointer as shown.



Here, the pointer variable **P2** contains the address of the pointer variable **P1**, which points to the location that contains the desired value. This is known as multiple indirections.

A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

int **P2;

This declaration tells the compiler that **P2** is pointer to a pointer of int type. Remember, the pointer **P2** is not a pointer to an integer, but rather a pointer to an integer pointer.

We can access the target value indirectly pointed to a pointer by applying the indirection operator twice. Consider the following code:

```
main ( )
{       int x,*p1,**p2;
        x=100;
        p1=&x;          /* address of x */
        p2=&p1;         /*address of p1*/
        printf("%d",**p2);
}
```

This code will display the value 100. Here, **p1** is declared as a pointer to an integer and **p2** as a pointer to a pointer to an integer.

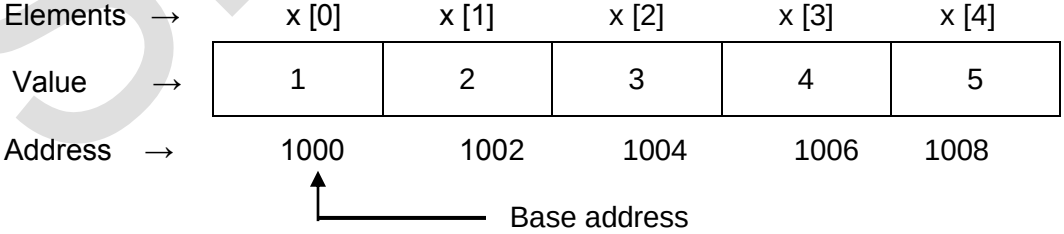
* POINTERS AND ARRAYS (POINTERS TO ARRAYS)

ARRAY NAME AS A POINTER

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in continuous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.
Suppose we declare an array x as follows:

int [x] = {1, 2, 3, 4, 5};

Suppose the base address of x is1000 and assuming that each integer requires two bytes, the five elements will be stored as follows:



The name **x** is defined as a constant pointer pointing to the first element, **x [0]** and therefore the value of **x** is 1000, the location where **x [0]** is stored. That is,

x = &x [0] =1000

If we declare **p** as an integer pointer, then we can make the pointer **p** to point to the array **x** by the following assignment:

p=x;

This is equivalent to **p = &x [0];**

Now we can access every value of x using p++ to move from one element to another. The relationship between **p** and **x** is shown as:

p= &x [0] (=1000)	p+3=&x [3] (=1006)
p+1=&x [1] (=1002)	p+4=&x [4] (=1008)
p+2=&x [2] (=1004)	

Address of an element is calculated using its index and the scale factor of the data type. For instance,

Address of **x [3]** = base address + (3 x scale factor of **int**)
=1000 + (3 x 2)

When handling arrays, instead of using array indexing, we can use pointers to access array elements. Note that ***(p+3)** gives the value of **x [3]**.The pointer accessing method is much faster than array indexing.

Example 5 illustrates the use of pointer accessing method.

Example-5 Write a program using pointers to compute the sum of all elements stored in an array.

The program shown in figure 8 illustrates how a pointer can be used to traverse an array element. By incrementing an array pointer it point to the next element, we need only to add one to **p** each time we go through the loop.

Program

```
main()
{ int *p,sum,i;
  int x[5] = {5,9,6,3,7};
  i = 0;
  p = x;    /* initialization 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

Accessing one-dimensional array elements using the pointer

*POINTERS AS FUNCTION ARGUMENTS

We can pass the address of a variable as an argument to a function in the normal fashion. 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 then address of the variable x is passed into the function change(). We are not passing value of x. In change() function, 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'**. Here **p** is the address of x. So the value of x is changed from 20 to 30. And 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"

Example-14 Write a function using pointers to exchange the values stored in two locations in the memory.

The program in figure 16 shows how the contents of two locations can be exchanged using their address locations. The function **exchange ()** receives the address 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 16 passing of pointers as function parameters

RETURNING MULTIPLE VALUES TO FUNCTIONS THROUGH POINTERS

return statement can return only one value to calling function. But we can get more information from a user defined function. Using arguments we can receive information in function and also send back information to calling function.

The arguments that are used to "send out" information are called **output parameters**.

Address operator (&) and **indirection operator (*)** are used in mechanism of sending more information back. E.g.

```
void sum_diff (int x, int y, int  *s, int *d);
void main()
{
        int x = 20, y = 10, s, d;
        sum_diff (x, y, &s, &d);
}

void sum_diff (int a, int b, int *sum, int *diff)
{
        *sum = a + b;
        *diff = a – b;
}
```

Actual arguments x and y are input arguments, s and d are output arguments. In function call, we pass actual values of x and y but we pass addresses of locations where the values of s and d are stored in memory.

Indirection operator * in sum and diff indicates that these variable used to store addresses and not to store values of variables.

*sum = a + b; will add a and b and store the result in memory location pointed by sum.

Variables *sum and *diff are known as pointers and sum and diff are known as pointer variables. They are declared as int so they can point to locations of int type data.

The use of pointer variables as actual parameters for communicating data between functions is called **"pass by pointers"** or **"call by address"** or **"call by reference"**.

*DYNAMIC MEMORY ALLOCATION (STORAGE MEMORY)

Introduction

We face situations in programming where the data is dynamic in nature. That is, the number of data items keeps changing during execution of the program.

For example, consider a program for processing the list of customers. The list grows when names are added and when names are deleted. When list grows we need to allocate more memory space to the list to accommodate additional data items.

Such situations can be handled more easily and effectively by dynamic data structures in conjunction with dynamic memory management techniques.

Dynamic data structures provide flexibility in adding, deleting or rearranging data items at run time. Dynamic memory management techniques permit us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space.

Dynamic Memory Allocation

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgment of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign the memory space during execution which is required by the variables in a program.

The process of allocating memory at run time is known as dynamic memory allocation. C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution.

These functions help us build complex application programs that use the available memory intelligently.

Memory Management Functions

Function	Task
malloc	Allocates request size of bytes and returns a pointer to the first byte of the allocated space
calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free	Frees previously allocated space
realloc	Modifies the size of previously allocated space

Allocating a block of memory: malloc

A block of memory may be allocated using the function **malloc.** The **malloc** function reserves a block of memory of specified size and returns a pointer of type **void.** This means that we can assign it to any type of pointer. It takes the following form:

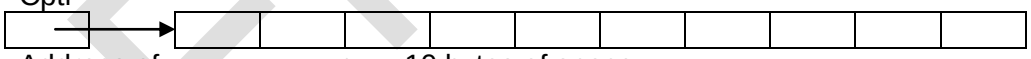
ptr = (cast-type *) malloc(byte-size);

ptr is a pointer of type cast-type. The malloc returns a pointer (of cast type) to an area of memory with size byte-size.

x = (int *) malloc (100 *sizeof(int));

Successful execution of this statement, a memory space equivalent to "100 times the size of an int" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type of int.

Similarly, the statement **cptr = (char*) malloc(10);** allocates 10 bytes of space for the pointer cptr of type char. This is illustrated as:



Storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer.

We can use malloc to allocate space for complex data types such as structures. Example:

st_var = (struct store *) malloc(sizeof(struct store));

where st_var is a pointer of type struct store.

The malloc allocates a block of contiguous bytes. The allocation can fail if the space in heap (free memory) is not sufficient to satisfy the request. If it fails, it returns a NULL. We should therefore check whether the allocation is successful before using the memory pointer.

Example: Write a program that uses a table of integers whose size will be specified interactively at run time.

Following program tests for availability of memory space of required size. If it is available then the required space is allocated and the address of the first byte of the space allocated displayed. The program also illustrates the use of pointer variable for storing and accessing the values.

```
#include <stdio. h>
#linclude <stdlib. h>
#include <alloc.h>
#define NULL 0

main ()
{
        int *p, *table;
        int size;
        printf("\nWhat is the size of table?");
        scanf ( "%d " , size) ;
        printf("\n")
        /* ----------Memory allocation ----------*/
        if ( (table = (int*)malloc(size *sizeof(int)) ) == NULL)
          {
              printf("No space available \n");
              exit(1);
          }
        printf("\n Address of the first byte is %u\n",table);
        /* Reading table values */
        printf("\n Input table values\n");
        for (p=table; p<table + size; p++)
            scanf("%d",p);
       /* Printing table values in reverse order  */
       for (p = table + size -1; p >= table; p --)
            printf("%d is stored at address %u \n",*p,p);
}
```

Output

```
What is the size of the table? 5
Address of the first byte is 2262
Input table values
11 12 13 14 15
15 is stored at address 2270
14 is stored at address 2268
13 is stored at address 2266
12 is stored at address 2264
11 is stored at address 2262
```

Allocating multiple blocks of memory: calloc

calloc is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures.

While malloc allocates a single block of storage space, calloc allocates multiple blocks of storage, each of the same size, and then sets all bytes to zero. The general form of calloc is:

ptr = (cast-type *) calloc (n, elem-size);

Above statement allocates contiguous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero and a pointer to the first byte of the allocated region is returned. If there is not enough space, a NULL pointer is returned.

Following segment of a program allocates space for a structure variable:

```
. . . .
struct student
{       char name[25];
        float age;
        long int id_num;
} ;                     .
typedef struct student record;
record  *st_ptr;
int class_size = 30;
st_ptr=(record *)calloc(class_size, sizeof(record));
. . . .
```

record is of type struct student having three members: name, age and id_num. The calloc allocates memory to hold data for 30 such records. We must be sure that the requested memory has been allocated successfully before using the st_ptr. This may be done as follows:

```
if (st_ptr == NULL)
{
        printf("Available memory not sufficient");
        exit(1);
}
```

Releasing the used space: free

Compile-time storage of a variable is allocated and released by the system in accordance with its storage class.

With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.

When we no longer need the data we stored in a block of memory, and we do not intend to use that for storing any other information, we may release that block of memory for future use, using the free function:
free (ptr);

ptr is a pointer to a memory block which has already been created by malloc or calloc. Use of an invalid pointer in the call may create problems and cause system crash. We should remember two things here:

1. It is not the pointer that is being released but rather what it points to.
2. To release an array of memory that was allocated by calloc we need only to release the pointer once. It is an error to attempt to release elements individually.

Note:
The calloc( ) functions works exactly similar to malloc( ) except for the fact that it needs two arguments. For example,
int *p,
p = (int *) calloc (10, 2) ;
Here 2 indicates that we wish to allocate memory for storing integers, (since an integer is a 2-byte entity) and 10 indicates that we want to reserve space for storing 10 integers.

Another minor difference between malloc( ) and calloc( ) is that, by default, the memory allocated by malloc( ) contains garbage values, whereas that allocated by calloc( ) contains all zeros. While using these functions it is necessary to include the file "alloc.h" at the beginning of the program.

Altering the size of a block: realloc

If is possible that the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it.

In both the cases, we can change the memory size already allocated with the help of the function realloc. This process is called the reallocation of memory. For example, if the original allocation is done by the statement

ptr = malloc(size);

then reallocation of space may be done by the statement

ptr = realloc(ptr, newsize);

This function allocates a new memory space of size newsize to the pointer variable ptr and returns a pointer to the first byte of the new memory block. The newsize may be larger or smaller than the “size”.

Remember, the new memory block may or may not begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.
If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost). This implies that it is necessary to test the success of operation before proceeding further.

Example: Write a program to store a character string in a block of memory space created by **malloc** and then modify the same to store a large string.

The output illustrates that the original buffer size obtained is modified to contain a larger string. Note that the original contents of the buffer remain same even after modification of the original size.

```
#include <stdio.h>
#include<stdlib.h>
#define <alloc.h>
#define NULL 0
main ()
{
    char *buffer;
    /* Allocating memory */
    if ( (buffer = (char *)malloc(10)) == NULL)
          {
            printf("malloc failed.\n");
            exit(1);
          }
    printf("Buffer of size %d created \n",_msize(buffer));
    strcpy(buffer,"HYDERABAD");
    printf("\nBuffer contains: %s \n",buffer);
    /* Reallocation */
    if ( (buffer = (char *)realloc(buffer,15)) == NULL)
     {  printf("Reallocation failed. \n");
        exit(1);
     }
    printf("\nBuffer size modified. \n");
    printf("\nBuffer still contains: %s \n",buffer);
    strcpy(buffer, "SECUNDERABAD");
    printf("\nBuffer now contains: %s \n",buffer);
    /* Freeing memory * /
    free(buffer) ;
}
```

Output

```
Buffer of size 10 created
Buffer contains: HYDERABAD
```

```
Bluffer size modified
Buffer still contains: HYDERABAD
Buffer now contains: SECUNDERABAD
```

Disclaimer

The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.