C-08 Pointers in C

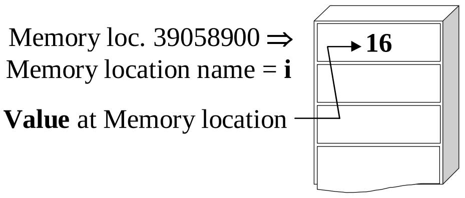
8.1 The Address Operator &:

Till now we have dealt with variable types that deal with integers, floating point numbers, characters etc. Apart from these, there is a special type of variable in C called **pointers** that can be used **to store the memory address locations of other type of variables**. Pointers are used extensively in C and the clever use of pointers has made C such a versatile and popular language. We will see in later sections how pointers can be used to efficiently handle strings and declare variables dynamically during runtime.

Consider the declaration: **`int i =16;`**

This declaration tells the C compiler to do the following things:

1. **Reserve space** in the memory to hold the integer value
2. **Associate the variable** name i with the particular memory location
3. Finally **store the value 16** in that memory location



Memory locations are discrete locations like the addresses of buildings and hence have discrete integral values like 1, 2, 3, … etc. Thus the memory location where the variable named i is stored has an integer value. In C there is a way to get the value of the memory location where a particular variable is stored. C provides us with a special operator called the ***address of*** operator to achieve this. The operator is represented by the ampersand symbol ***&***. The following example shows a method to **view the address location** where the variable i is stored:

```
main()
{  int i = 16;
   printf(“\nAddress of i = %u”, &i );
   printf(“\nThe value of i = %d”, i );
   return (0);
}
```

Output of the above program is:

```
   Address of i = 39058900
   The value of i = 16
```

The expression **`&i`** returns the address where the value of the variable **`i`** is stored. In this case it is found to be the memory location number **`39058900`**. Since an address location is always a positive integer, we have used the format specifier **`%u`** to indicate an unsigned integer type variable.

Note: For the ***address of*** operator **`&`** to act, a variable on which it acts should have a unique address location. Thus it cannot work on the result of any arithmetic operation like **`&(5*(x+y));`**

8.2 The Indirection Operator ‘ * ’:

To display or access the value contained in a particular memory location, one can simply use the variable name as had been done till now. However there is another way to indirectly access the value of a variable using a special operator called the **indirection operator** *. The indirection operator acts on address operands and displays the value contained in that particular address location. Thus the indirection operator can also be called the ***value at address operator pointed to by the pointer***. The following example shows the use of the indirection operator to access the value contained in a variable.

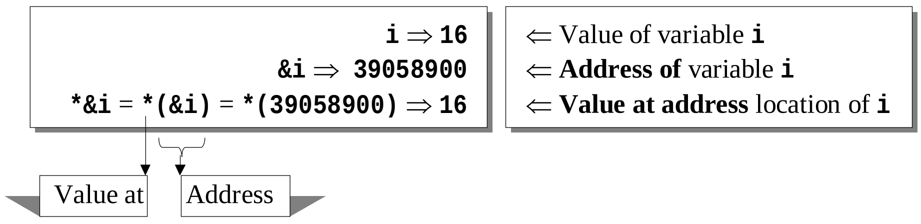
```
main()
{  int i = 16;
   printf(“\nAddress of i = %u”, &i );
   printf(“\nThe value of i = %d”, i );
   printf(“\nThe value of i = %d”, *(&i) );
   return (0);
}
```

Output of the above program is:

```
   Address of i = 39058900
   The value of i = 16
   The value of i = 16
```

Since **&i** represents the address where value of the variable **i** is stored, the * or indirection operator when placed before the address i.e. before **&i** will give the value stored at the address location **&i**. In the above case, the value stored at the address location of **i** is **16** as shown by the program.

Thus:



When we had run the above program, we got the address location for the variable **i** as **39058900**. However **this is not a fixed value** and will change from computer to computer and within the same computer for each run of the program. This is because, each time a variable is declared, it is assigned a block of address in the memory. This address space is **not a fixed one** and every time the variable is declared, the compiler assigns a separate address space for the variable. From this we can conclude that the **address of a variable is itself a variable** number and hence can be stored in another variable.

8.3 How to declare Pointers:

In the previous section we have seen that the address location of a variable is itself a variable quantity. Till now we have only displayed the address location. To store this **variable address location** we need a special type of variable called a **pointer variable**. Being a special type of variable, a pointer variable, or simply a pointer, needs to be declared in a special way too.

To declare a pointer variable, we have to first know the type of the variable whose address the pointer will store, i.e. whether the pointer will store the address of an integer or a float or character etc. The type of the pointer will be the same as the type of the variable whose address it will store. Finally we declare the pointer variable by placing a * **in front of the variable name** after stating the type of the pointer. The following examples show declarations of different types of pointers:

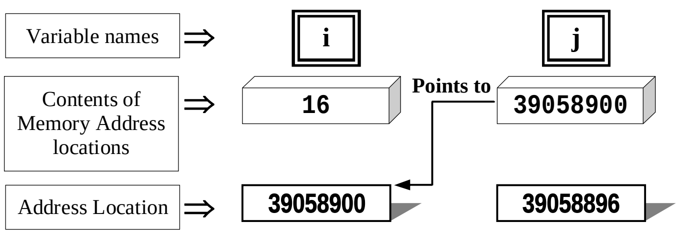
```
int *j;     → declares a pointer variable called  j which can hold the address of an integer type variable
float *k;   → declares a pointer variable called  k which can hold the address of a float type variable
double *l;  → declares a pointer variable called  l which can hold the address of a double type variable
char *m;    → declares a pointer variable called  m which can hold the address of a character type variable
```

The following program piece illustrates the use of pointers to address and display variables:

```
main()
{  int i;
   int *j;                                  → declares an integer type pointer variable called  j
   j=&i;                                    → assigns the address of the variable i to j
   i=16;
   printf("\nAddress of i = %u", &i );      → prints the address of i using the address operator &
   printf("\nAddress of i = %u", j );       → prints the address of i stored in the pointer j
   printf("\nValue of i = %d", i );         → prints the value of i using the variable itself
   printf("\nValue of i = %d", *&i );       → prints the value of i using the indirection operator
   printf("\nValue of i = %d", *j );        → prints the value of i using the indirection operator
   return (0);                                on the pointer  j
}
```

Output of the above program:

```
   Address of i = 39058900
   Address of i = 39058900
   Value of i = 16
   Value of i = 16
   Value of i = 16
```



The diagram shows the relation between the different variables, and the values stored in those variables. It can be seen that the memory location indicated by the variable name **i** has the physical address **39058900**. Since a pointer is also a variable, it also needs to be stored in a particular memory location. From the diagram we can see that the memory location indicated by the variable name **j** has the physical address **39058896**. Thus whatever values are assigned to the two variables **i** and **j** during program execution, will get stored in the memory address locations **39058900** and **39058896** respectively.

Since the value **16** is assigned to the variable **i** in the program, this value gets stored in the memory location **39058900**. Similarly, the address of the variable **i** is assigned to the pointer variable **j** and **hence this address** i.e. the value **39058900** gets **stored in the memory location 39058896**.

From the above examples it may seem that a pointer when declared can be used to hold the address of a particular variable only. However a **pointer being a variable** can be assigned **different addresses** at different times of program execution. In the following example, a single float type pointer is used to hold the address of more than one variable at different points of program execution.

```
main()
{  float m=2.5, n;                       → declares two float type variables called m and n
   float *j;                             → declares a float type pointer variable called j
   j = &m;                               → assigns the address of the variable m to j
   printf(“\nAddress of m = %u”, j );    → prints the address of m stored in the pointer j
   printf(“\nValue of m = %.2f”, *j );   → prints the value of m using the indirection operator
   n = m*m;
   j = &n    ;                           → assigns the address of the variable n to j
   printf(“\nAddress of n = %u”, j );    → prints the address of n stored in the pointer j
   printf(“\nValue of n = %.2f”, *j );   → prints the value of n using the indirection operator
   return (0);
}
```

Output of the above program:

```
   Address of m = 39058900
   Value of m = 2.50
   Address of n = 39058896
   Value of n = 6.25
```

The above program has not used the variable names directly to display the contents of the variables **m** and **n**. Instead the program makes use of the indirection operator to indirectly display the contents of the memory locations stored in the pointer **j**. In the first case the pointer points to the memory location of **m** and hence prints the value of **2.50**. Then the memory location of **n** is assigned to **j** and hence in the second case the contents of the memory location pointed to by **j** becomes **2.50*****2.05**=**6.25**. Also note that the memory blocks where **m** and **n** are stored are different, as indicated by the contents of the memory locations in **j**.

Unlike other unary operators, an **indirection operator can even be used at the left of an assignment operator** as shown in the example below. This is possible as the indirection operator refers to the value at the address location indicated by the pointer.

```
main()
{  int num, *p_num;                                → integer type pointer p_num declared
   p_num=&num;                                     → p_num assigned to address of num
   printf(“\nEnter a number: ”);
   scanf(“%d”, p_num);                             → value assigned to &num i.e. p_num
   printf(“\nEntered number = %d”, *p_num);        → value at address pointed to by p_num
   *p_num=num*num;                                 → num*num assigned to contents of
                                                     memory location pointed to by p_num
   printf(“\nSquare of the number=%d”, *p_num);    → value at address pointed to by p_num
   return (0);
}
```

Output of the above program:

```
   Enter a number: 5
   Entered number = 5
   Square of the number=25
```

In the above piece of program three things need to be noted. First we have directly used the pointer `p_num` instead of `&num` (which in reality indicates the address location of num as does `p_num`). Secondly we have printed the values of `num` indirectly by using the indirection operator on the pointer `p_num`. Thirdly, we have assigned the product `num*num` indirectly to the contents of the address location pointed to by `p_num`.

Thus: `*p_num=num*num;` is **same as**
`num=num*num;` which in turn is **same as**
`*p_num=(*p_num)*(*p_num);`

8.4 How to initialise a Pointer:

Just like any other variable, a pointer variable can also be initialised during declaration. An un-initialised pointer can point to anywhere in the memory and can accidentally write to area of memory that holds other data.

```
int num;
int *p_num=&num;
```

In the above declaration, an integer pointer is declared called `p_num` and assigned the memory address of the integer variable `num`. However before initialising a pointer with the address of another variable, the **variable must be declared prior to the pointer declaration**.

A pointer can also be initialised during its declaration without the address of any specific variable, by using the pointer equivalent of a zero, i.e. a `NULL` pointer.

```
int *p_num=NULL;
```

The above statement indicates that it is not pointing to any address location. A `NULL` pointer can be used to check memory availability during dynamic memory allocations while using the memory allocation functions malloc`()`, `calloc()`. This is **essential to avoid any runtime error** in case sufficient memory is not available as requested. The following line of code illustrates how the condition is checked:

```
if(p_num==NULL) printf(“\nMemory allocation was not possible”);
```

8.5 Pointers as Function Arguments:

We had seen that when a variable is passed to a function, a copy of the value of the variable is passed to the function as a formal argument. Thus whatever changes are made to the formal argument inside the function, does not affect the actual argument. However there is another way to pass a variable to a function.

Instead of passing the actual argument if the **address of the variable is passed**, then whatever changes are made to the contents of that particular address location will permanently change the value at the address location. This type of passing an argument by its address is known as **pass by reference**, as opposed to pass by value, where a copy of the value of the argument is passed.

Passing an address of a variable to a function implies, we have to **pass a pointer** to the function, because the address of a variable can be stored in a pointer type variable only. The following two examples illustrate the difference between pass by value and pass by reference.

```
#include <stdio.h>
void exchange(int *p_a, int *p_b);

void main()
{  int num1, num2;
   printf(“\nEnter number1: ”); scanf(“%d”, &num1);
   printf(“\nEnter number2: ”); scanf(“%d”, &num2);
   printf(“\nContents of Number1=%d and Number2=%d”, num1, num2);

   exchange(&num1, &num2);

   printf(“\nContents of Number1=%d and Number2=%d”, num1, num2); }

void exchange(int *p_a, int *p_b)
{  int temp;
   temp=*p_a;     → contents of memory location pointed to by p_a assigned to temp
   *p_a=*p_b;     → contents of memory location pointed by p_b assigned to location pointed by p_a
   *p_b=temp;     → contents of temp assigned to memory location pointed to by p_a
}
```

Output of the above program:

```
Enter number1: 2
Enter number2: 200

Contents of Number1 = 2 and Number2 = 200

Contents of Number1 = 200 and Number2 = 2
```

The function **`exchange()`** is **called by reference** with the address of the two variables **`num1`** and **`num2`**. The address is received by the two pointer variables **`p_a`** and **`p_b`** respectively defined within the function header. Inside the function, the content of the memory location pointed to by **`p_a`** (i.e. **`*p_a`**) is assigned to the temporary variable **`temp`**. Then the content of the memory location pointed to by **`p_b`** is assigned to the memory location pointed to by **`p_a`**. Finally the contents of **`temp`** is assigned to the memory location pointed to by **`p_b`**. Thus the actual contents of the memory locations pointed to by **`p_a`** and **`p_b`** are exchanged as reflected in the last **`printf()`** statement which prints the exchanged values for the same order of the variables.

The next program illustrates the use of pointers in functions by **calling two separate functions – one by value and one by reference** – to do the same calculation, and see what difference does it make.

```
#include <stdio.h>
int square_by_val(int a);
int square_by_ref(int *p_a);

void main()
{  int num, sq, *p_num;
   p_num=&num;

   printf(“\nEnter a number: ”);
   scanf(“%d”, p_num);

   sq=square_by_val(num);                  → function call by value with num as argument

   printf(“\nEntered number = %d”, num);
   printf(“\nSquare of number = %d”, sq);  → value of sq as returned by the function, printed

   sq=square_by_ref(&num);                 → function call by reference with address of num

   printf(“\nEntered number =%d”, num);
   printf(“\nSquare of number =%d”, sq);   → value of sq as returned by the function, printed
}

int square_by_val(int a)                   → formal argument is the integer variable a
{  a=a*a;                                  → new value of formal argument = a*a
   return a;                               → new value of formal argument returned
}

int square_by_ref(int *p_a)                → formal argument is the integer pointer p_a
{  *p_a=(*p_a)*(*p_a);                     → contents of memory location p_a changed
   return *p_a;                            → new contents of memory location p_a returned
}
```

Output of the above program:

```
Enter a number: 6

Entered number = 6
Square of number = 36

Entered number = 36
Square of number = 36
```

In the above example, we have used two separate functions to calculate the square of a number. The first function is a **call by value** function **`square_by_val()`**, with the actual argument as the variable name **`num`**. The function receives a **copy of the value** of the actual argument inside its formal argument **`a`**. Finally the square of **`a`** is calculated and the new value assigned to **`a`** and returned by the function. This part is the same as we had learned with functions. Next, within the **`main()`** function the unchanged actual argument **`6`** is printed followed by the value returned by the function i.e. **`36`**.

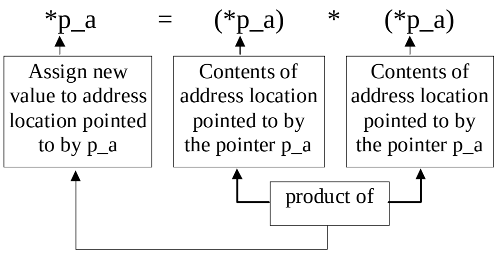
However the next function call **`square_by_ref()`** is a **call by reference** i.e. we pass the **address of the variable** and not a copy of the value of the variable. Let us now see how this affects the output.

First notice the function call. Instead of the argument **`num`**, we have passed the address of the variable **`num`** i.e. **`&num`**. For the formal argument to receive this address, it has to be a pointer variable, as only a pointer variable can hold an address. Thus the function header contains the formal argument defined as a pointer:

`int square_by_ref(int *p_a)` → **`int *p_a`** defines an integer type pointer **`p_a`**

Argument is a pointer that receives the address of an **`int`** type variable

Finally in the body of the function **`square_by_ref()`**, we are changing the contents of the address location received by the pointer **`p_a`** (i.e. the value **`*p_a`**) through the statement **`*p_a=(*p_a)*(*p_a)`**.


*p_a = (*p_a) * (*p_a)
Assign new value to address location pointed to by p_a
Contents of address location pointed to by the pointer p_a
Contents of address location pointed to by the pointer p_a
product of

The value in the memory location **`p_a`** as entered by the user (i.e. *p_a, which in our case is **`6`**) is first taken to the accumulator and the product **`6x6`** calculated. The result is then stored back into the address location pointed to by **`p_a`** (i.e. **`*p_a`**). This results in a change in the actual variable value ‘**`6`**’ stored in the memory location **`&num`** to **`6x6=36`**. Thus after the operation, the memory location **`p_a`** contains the value **`36`** and not **`6`**. This makes the last two **`printf()`** statements print **both** the entered number and the square number as 36 as the actual value at the memory location for **`num`** has been changed from **`6`** to **`36`**.

Hence whenever a **pass by reference** occurs, the value stored at the actual argument memory location gets accessed. Therefore any change made on the value thus affects the actual argument as well.

8.6 Pointers to Functions:

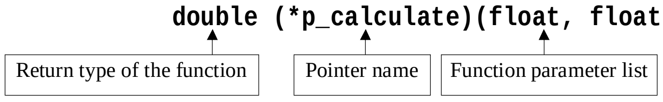
Till now we have used a pointer to point to a variable data type. C extends this idea to functions as well i.e. **a pointer can be declared to point to a function**. This is possible because when we define a function, the code for the function starts from a certain memory location. To call a function through a pointer, the pointer is assigned to this memory location. The particular type of function that will be called depends on the last function assigned to the pointer. Thus using the same pointer several functions can be accessed.

To declare a pointer to a function, the following things need to be specified along with the pointer name:

- The **return type** of the function
- The **parameter list** of the function

Let us now define a pointer called p_**`calculate`** to a function. The function for which the pointer is defined, takes two **`float`** type variables and returns a **`double`** type variable. The pointer is defined as:

```
double (*p_calculate)(float, float);
```



At first glance this may seem a little strange, with all sorts of brackets placed everywhere. But the pair of brackets surrounding the pointer name is necessary to differentiate it from a function declaration. The asteryx * placed before the pointer name **`p_calculate`** indicates it to be a pointer. The following examples are given to help the student differentiate between function and pointer declarations.

`double calculate(float, float);`	→ indicates a function declaration with the name of the function as **`calculate`**, which takes two **`float`** type variables and returns a **`double`** type variable
`double *p_calculate;`	→ indicates the declaration of a pointer called **`p_calculate`** which points to the memory location of a **`double`** type variable
`double *p_calculate(float, float);`	→ indicates a function declaration with the name of the function as **`p_calculate`**, which takes two **`float`** type variables and **returns** a **`double`** type **pointer** variable
`double (*p_calculate)(float, float);`	→ indicates the declaration of a pointer called **`p_calculate`** which points to the memory location of

a function which takes two **float** type variables and returns a **double** type variable

We will now use the above pointer to a function to write a calculator program.

```
#include <stdio.h>
double add(float a, float b);
double sub(float a, float b);
double mul(float a, float b);
double div(float a, float b);

int main()
{ int prompt=0;
  float num1, num2;

  double (*p_calculate)(float, float);

  printf("\nEnter first number: "); scanf("%f",&num1);
  printf("\nEnter second number: "); scanf("%f",&num2);

  puts("Enter \t<1> for Addition \n\t<2> for Subtraction");
  puts("\t<3> for Multiplication \n\t<4> for Division");
  scanf("%d",&prompt);

  switch(prompt)
   {case 1: p_calculate = add;
            break;
    case 2: p_calculate = sub;
            break;
    case 3: p_calculate = mul;
            break;
    case 4: p_calculate = div;
   }

  printf("\nThe result is %lf", p_calculate(num1, num2));
  return 0;
}

double add(float a, float b)
{ return (a+b); }

double sub(float a, float b)
{ return (a-b); }

double mul(float a, float b)
{ return (a*b); }

double div(float a, float b)
{ return (a/b); }
```

Output of the above program:

```
Enter first number: 10
Enter second number: 20

Enter   <1> for Addition
        <2> for Subtraction
        <3> for Multiplication
        <4> for Division
2

The result is -10.000000
```

The pointer to a function **p_calculate** can point to any function consisting of two arguments of type **float** and returning a value of type **double**. In the **main()** function two variables **num1** and **num2** are taken as inputs and based on the input **prompt**, the **switch-case** block assigns the corresponding functions to the pointer **p_calculate**.

One thing should be noted in the above example. When a function is assigned to a pointer, **only the function name without any brackets or argument list is assigned to it**. The function name is automatically converted to an address and stored in the pointer. Similarly **when calling a function** through a pointer, the **name of the pointer is used just like the function name**. The pointer name is followed by the argument list enclosed within a pair of brackets. Thus in the above case, the result of:

`p_calculate(num1, num2)` <u>**is the same as**</u> **`add(num1, num2)`** <u>**or**</u> **`sub(num1, num2)`** etc.

We have seen how we can declare a pointer to a function. Since a function can take a pointer to a variable as an argument similarly it can also take a pointer to a function as an argument. In this way a function can be passed to another function as an argument.

8.7 Pointers and Arrays:

Arrays and pointers work in a similar manner and one can interchange array and pointer notations. This is because the **name of a one-dimensional array gets automatically converted to a pointer** and the array name points to the zeroth element of the array.

To understand the working of arrays and pointers we have to first understand **pointer arithmetic**. One can do addition, subtraction and comparisons with pointers but **multiplication and division are not allowed**.

Let us consider an integer array and see how it is related to a pointer.

```
int marks[10];
```

Once the above array is declared, then the variable **`marks`**, **without any square brackets** refer to a pointer that points to the 0th element of the array **`marks[]`**. Thus:

`marks` <u>is same as</u> **`&marks[0]`**

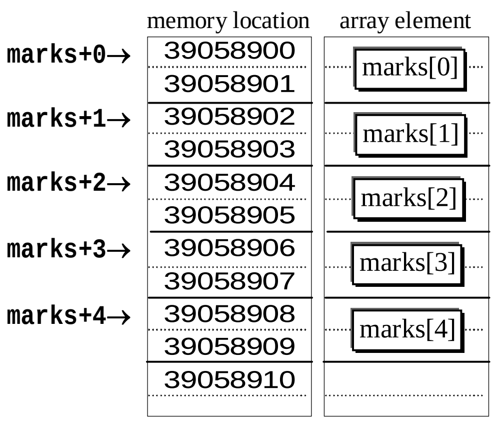
Let us now see how we can use arithmetic operations on pointers and what do they mean. We have seen that when an array name is declared, the name by itself acts as a pointer and points to the 0th element of the array. For the array **`marks[]`** if we write the statement,

```
marks = marks+1;
```

OR

```
marks++;
```

then as per our knowledge till now, the above statement would imply that if the pointer initially points to the memory location say **39058900**, then after the statement is executed, it will point to the location **39058901**. However <u>**this is incorrect**</u>. In reality it will point to the memory location **39058902**.



Why? Simply because of the fact that **pointers increment based on the data type they point to**. This means that since an **`int`** type data occupies two bytes, the pointer will increment by units of two bytes, to point to the next data element of the same type, with each increment. This is shown by the diagram above. This behaviour of pointers is utilised to move through an array, because when an array is declared, the array elements occupy consecutive memory locations. Hence **incrementing a pointer** will always **point to the next array element** (provided it is not the last array element) and **not to something in-between**.

In case we have float type data, the pointer will have to be declared as a **`float`** type pointer and the statement **`marks++`** will move the pointer by 4 bytes for every increment of the pointer. Thus if **`marks`** points to **38058200**, then after the execution of **`marks++`**, **`marks`** will point to the location **38058204** and **not to 38058201**. We can conclude that:

`marks`	<u>is same as</u>	**`&marks[0]`**	
`(marks+0)`	<u>is same as</u>	**`&marks[0]`**	
`(marks+1)`	<u>is same as</u>	**`&marks[1]`**	
`(marks+4)`	<u>is same as</u>	**`&marks[4]`**	<u>in general</u>,
`(marks+i)`	<u>is same as</u>	**`&marks[i]`**	

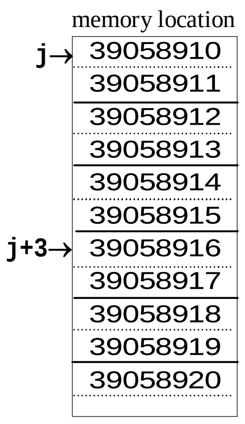
Since a pointer is a variable, we can perform the following operations on pointers:

- **Add a number** to a pointer
- **Subtract a number** from a pointer

- **Subtract one pointer** from another
- **Compare two pointer** variables

However following **operations are <u>not possible</u> on pointers** for obvious reasons:

- Add two pointers
- Multiply two pointers
- Multiply a pointer with a number
- Divide two pointers
- Divide a pointer with a number



```
int i, *j;
j=&i;
j=j+3;
```

The above statements will make the pointer **j** finally point to **6** memory locations ahead of location for **i**, since **i** being an integer will occupy 2 bytes and the statement **j=j+3** will move the pointer by (3 x 2-bytes per integer) = 6 memory locations. Thus if initially **j** points to the location **39058910**, then after the execution of the statement **j=j+3**, the pointer will point to the memory location **39058910+6=39058916** as shown in the diagram to the right.

Similarly, subtracting a number from a pointer will make the pointer point to a previous memory location.

However **to subtract** one pointer from another, it needs to be assured that **both the pointers point to elements of the same array**. The difference indicates the number of bytes separating the two array elements to which the two pointers point. The following example calculates the length of a word in a character array.

```
main()
{  char str[]={“the sound of music”};
   char *p_str1, *p_str2;
   int word_length;

   p_str1 = &str[4];      → pointer p_str1 points to the character s in the string array str[]
   p_str2 = &str[9];      → pointer p_str2  points to the blank character after d in the string array

   word_length = p_str2 - p_str1;
   printf(“\nWord Length=%d”, word_length);
   return 0;
}
```

Output of the above program:

```
       Word Length=5
```

We have seen that the array name **marks** points to the zeroth element of the array **marks[]**. Thus ***marks** indicates the contents of the zeroth position of the array, i.e. ***marks** ⇒ **marks[0]**. Now ***marks** is the same as ***(marks+0)**. Thus we can represent the 4th element of the array by the expression ***(marks+4)**. In general we can express the **i**th element of the array **marks[]** as ***(marks+i)**. Therefore:

```
*(marks+0) ⇒ marks[0]
*(marks+4) ⇒ marks[4]
*(marks+i) ⇒ marks[i]
```

The fact is that, whenever we write a statement lik**e marks[i]**, the C compiler immediately converts it using pointer notation to the equivalent form ***(marks+i)**. Hence the following statements are all same, as after they are converted to the above notation by the compiler they all represent the same thing:

```
➢ marks[i]       ⇒  *(marks + i)
➢ *(i + marks)   ⇒  *(marks + i) ⇒ marks[i]
➢ i[marks]       ⇒  *(i + marks) ⇒ *(marks + i) ⇒ marks[i]
```

∴ All these represent the **same thing**

Next let us see how we can use this technique of accessing array elements using pointers, to **manipulate arrays using pointers only**. The following program illustrates the above use by calculating the *standard deviation* of a set of numbers.

```
/*Program to use Pointers to manipulate an Integer Array*/
#include <stdio.h>
```

```
#include <math.h>
#define MAX 100

int main()
{ int num[MAX], i, count;
  int *j;
  float sum1=0.0, sum2=0.0, term, mean, sd;

  j=num;

  printf("\nEnter number of elements in the list (<=%d): ", MAX);
  scanf("%d", &count);

  for(i=0; i<count; i++)
     { printf("Enter number-%d: ",(i+1));
       scanf("%d", j);
       sum1=sum1 + (*j);
       j++;
     }

  mean=sum1/count;

  j=num;                    → Reassigns the pointer j to point to the 0th element of the array num[]

  for(i=0; i<count; i++)
     { term=(mean - (*j)) * (mean - (*j));
       sum2=sum2 + term;
       j++;
     }

  sd=sqrt(sum2/count);

  printf("\nThe required standard deviation is %.3f", sd);
  return (0);
}
```

Output of the above program:

```
Enter number of elements in the list (<=100): 5

Enter number-1: 2
Enter number-2: 6
Enter number-3: 3
Enter number-4: 5
Enter number-5: 8

The required standard deviation is 2.135
```

Thus pointers provide us with another way to traverse an array and access its elements. In the above example we had used the array within the same main function. However **we can pass an entire array to a function** using pointers as shown in the next example below.

```
#include <stdio.h>
#define MAX 100

void entry(int *list, int n);      → Function prototype with an integer pointer list
void display(int *list, int n);    → Function prototype with an integer pointer list

main()
{ int num[MAX], i, count;  → Integer array num[] defined
  printf("\nEnter number of elements in the list (<=%d): ", MAX);
  scanf("%d", &count);

  entry(num, count);       → Function entry() called with argument as num (not num[])
  display(num, count);     → Function display() called with argument as num (not num[])
  return (0);
}

void entry(int *list, int n)
{ int i;
  for(i=0; i<n; i++)
    {printf("Enter number%d: ",(i+1));
     scanf("%d", list);    → scanf() enters value into the memory position pointed to by list
     list++;                  → The pointer list is incremented to point to the next array element
    }
}
void display(int *list, int n)
{ int i;
  printf("\nThe entered list is:");
  for(i=0; i<n; i++)
    {printf("\nNumber%d: %d",(i+1), *list);     → *list indicates the value at the location
                                                   pointed to by list
     list++;                  → The pointer list is incremented to point to the next array element
    }
}
```

Output of the above program:

```
Enter number of elements in the list (<=100): 5
Enter number1: 6
Enter number2: 2
Enter number3: 3
Enter number4: 5
Enter number5: 8

The entered list is:
Number1: 6
Number2: 2
Number3: 3
Number4: 5
Number5: 8
```

8.8 How to declare Pointer to Pointers:

We have seen that a pointer can be used to store variable address values i.e. a pointer by itself is a variable. This implies that just as a pointer variable is used to hold the address of another variable, in a similar manner a pointer can be declared to hold the address of another pointer variable. In case you find the above sentence confusing, read it more than once to make the meaning clear.

Such a pointer is called a **pointer to a pointer**, which holds the address of another pointer. To declare a pointer to a pointer we use the same convention i.e. we use an * to indicate a pointer variable. However since this pointer variable points to another pointer variable, instead of using a single * we use a pair of **. Thus:

`char **k;` → indicates a **pointer to a pointer variable** which holds the address of a `char` type variable
`int **k;` → indicates a **pointer to a pointer variable** which holds the address of a `int` type variable
`float **k;` → indicates a **pointer to a pointer variable** which holds the address of a `float` type variable

This idea can be extended further to declare a pointer to a pointer to a pointer and so on and so forth and with each step an extra * should be placed before the variable name. Thus `int ***k` indicates a pointer variable which stores the address of another pointer variable which stores the address of another pointer variable which in turn stores the address of an `int` type variable. The following example shows the use of a pointer to a pointer.

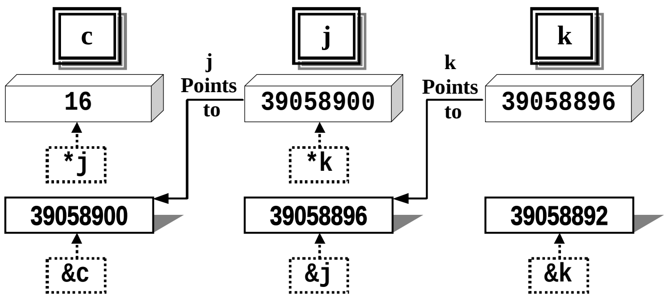
```
main()
{  int c=16;                            → declares an integer type variable called c
   int *j;                              → declares an integer type pointer called j
   int **k;                             → declares an integer type pointer to a pointer called k
   j=&c;                                → assigns the address of the variable c to j
   k=&j;                                → assigns the address of the pointer variable j to k
   printf("\nAddress of c = %u", &c );  → prints the address of c using the address operator &
   printf("\nAddress of c = %u", j );   → prints the address of c stored in the pointer j
   printf("\nAddress of c = %u", *k );  → prints the address of c stored in the pointer j

   printf("\nValue of c = %d", c );     → prints the value of c using the variable itself
   printf("\nValue of c = %d", *&c );   → prints the value of c using the indirection operator
   printf("\nValue of c = %d", *j );    → prints the value using the indirection operator on j
   printf("\nValue of c = %d", **k );
```
→ prints the value using the indirection operator on k

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

Output of the above program:

```
   Address of c = 39058900
   Address of c = 39058900
   Address of c = 39058900
   Value of c = 16
   Value of c = 16
   Value of c = 16
   Value of c = 16
```



The program prints the address of the variable **`c`** using the first three **`printf()`** statements in three different ways. All these produce the same value **`39058900.`** The meaning of each statement is explained below:

`&c` → The **Address of** operator **&** acts on the variable **`c`** to give the address of **`c`**.

`j` → As **`j=&c`**, the pointer **`j`** directly gives the address of **`c`**.

`*k` → The **Indirection** operator **`*`** is used to get the **<u>contents of the memory location pointed to by `k`</u>**. As can be seen from the diagram above, the memory location pointed to by **`k`** is **`39058896`**. The content of this memory location is **`39058900`**, which is the memory location of **`c`**.

Similarly the last four **`printf()`** statements are used to print the value contained in c in four different ways. The meaning of each statement is explained below:

`c` → The value in c printed directly.

`*&c` → The contents of the address location of c [**`*(&c)`**].

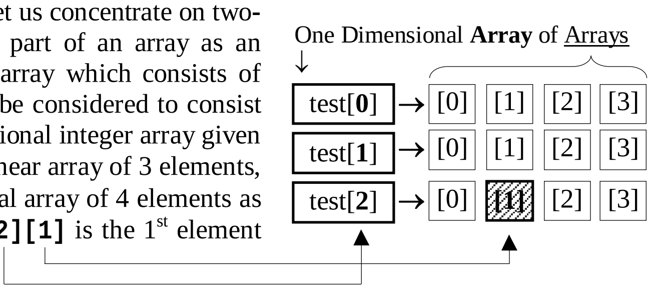
*j → The contents of the memory location pointed to by the pointer **`j`**. As **`j`** points to the memory location **`39058900`** whose contents are **`16`**, the same is printed**.**

`**k` → Now **`*k`** indicates the contents of the memory location pointed to by **`k`**. In this case it is **`39058900`**. Thus **`*(*k)`** indicates the contents of the location **`(*k)`**. In this case the contents of the location **`39058900`** is **`16`**. This can be explained in another way as shown below:

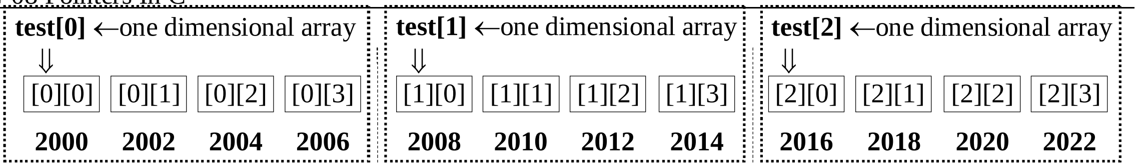
`**k   = **(&j)`	As **`k=&j`**
`= *(*&j)`	
`= *(&c)`	As **`*&j =`** contents of the address location of **`j`** = address of **`c`** = **`&c`**
`= c`	As **`*&c =`** contents of the address location of **`c`** = **`c`** = **`16`**

8.9 Pointers and Multi-dimensional Arrays:

When dealing with multi-dimensional arrays, let us concentrate on two-dimensional arrays. In C, we can consider a part of an array as an individual array. Thus in a two dimensional array which consists of rows and columns, each row of the array can be considered to consist of a one-dimensional array. Thus a two dimensional integer array given by **`int test[3][4]`** can be thought of as a linear array of 3 elements, where each of the elements is a one-dimensional array of 4 elements as sown in the diagram to the right. Thus **`test[2][1]`** is the 1st element of the 2nd row, as shown by the shaded box.


One Dimensional **Array** of <u>Arrays</u>
test[0] → [0] [1] [2] [3]
test[1] → [0] [1] [2] [3]
test[2] → [0] [1] [2] [3]

We had seen earlier that for a one dimensional array, say **`int age[5]`**, the array name **`age`** itself indicates a pointer to the 0th element of the array i.e. **`age`** is same as **`&age[0]`** which in turn is same as **`(age+0)`**. Simlarly **`(age+2)`** indicates the address of the second element of the array, i.e. it is same as **`&age[2]`**.



Therefore if we consider each row of the two dimensional array `test` as an individual one dimensional array, then we can consider `test[0]`, `test[1]`, `test[2]` etc. as the individual names of the one dimensional arrays. We had seen that an array name by itself represents a pointer to the 0th element of that array. **Using this interpretation, for a two dimensional array the notation with a single index like `test[i]` therefore represents a pointer that points to the 0th element of the one dimensional array `test[i]`**. Similarly `test[1]` represents a pointer that points to the 0th element of the one dimensional array `test[1]` etc.

Now if we consider the one dimensional array `test` with array elements `test[0]`, `test[1]`, … `test[2]`, we had seen that by itself the array name `test` or `(test+0)` represents a pointer to the first element of the array i.e. to `test[0]`, `(test+1)` represents a pointer to the 1st element of the array i.e. to `test[1]` etc.

Therefore from the above memory map we can see that for a **two dimensional array**, `test` or `test[0]` or `(test+0)` indicates the memory location `2000`. Similarly `test[1]` or `(test+1)` will indicate the memory location `2008` (and **not** `2002`), `test[2]` or `(test+2)` will indicate the memory location `2016` (and **not** `2004`). Therefore each individual row of the two dimensional array can be accessed by the notations `(test+0)`, `(test+1)`, and `(test+2)`.

Now we will see how to access each individual element of each row using pointers. We had seen that `test[1]` or `test+1` is interpreted by the compiler as the address location `2008` i.e. as the address of the first one dimensional array `test[1]`. Now `test[1]` being a one dimensional array, `(test[1]+1)` will give the address of the first element of that array i.e. will point to the address location `2010`. The value at this address location can be obtained by using the indirection operator i.e. by `*(test[1]+1)`.

However while studying one dimensional array we had seen that `test[i]` is same as `*(test+i)`. Replacing this notation in the above expression we get:

`*(test[1]+1)` ⇒ `*(*(test+1)+1)`

`*(test[2]+3)` ⇒ `*(*(test+2)+3)` etc.

In general `test[i][k]` ⇒ `*(test[i]+k)` ⇒ `*(*(test+i)+k)`

The following program example **tests whether a square matrix is a unitary matrix**, i.e. whether only the diagonal elements have value equal to 1, whereas all other elements are 0.

```
#include <stdio.h>
#define MAX 3

int main()
{ int matrix[MAX][MAX], i, j, flag=0;
  printf("Enter the elements of the %dx%d Matrix\n\n", MAX, MAX);

  /*LOOP TO INPUT THE MATRIX VALUES*/
  for(i=0; i<MAX; i++)
     for(j=0; j<MAX; j++)
        {printf("Enter element[%d][%d]: ", i+1, j+1);
         scanf("%d", (*(matrix+i)+j) );
        }

  /*LOOP TO DETERMINE IF MATRIX IS UNITARY*/
  for(i=0; i<MAX; i++)
     for(j=0; j<MAX; j++)
        {if(i!=j)
           {if( *(*(matrix+i)+j) != 0 )
              flag = 1;
           }
         else
           {if( *(*(matrix+i)+j) != 1 )
              flag = 1;
           }
        }
```

```
 /*LOOP TO DISPLAY THE MATRIX*/
 for(i=0; i<MAX; i++)
    {for(j=0; j<MAX; j++)
        printf(" %d\t", *(*(matrix+i)+j) );
     putchar('\n');
    }

 if(flag==0) puts("\nMatrix is Unitary");
 else        puts("\nMatrix is not Unitary");

 return 0;
}
```

Output of the above Program:

```
Enter the elements of the 3x3 Matrix

Enter element[1][1]: 1
Enter element[1][2]: 0
Enter element[1][3]: 0
Enter element[2][1]: 0
Enter element[2][2]: 1
Enter element[2][3]: 0
Enter element[3][1]: 0
Enter element[3][2]: 0
Enter element[3][3]: 1

 1       0       0
 0       1       0
 0       0       1

Matrix is Unitary
```

The matrix values are input through the **`scanf()`** statement. The variables to which the values are input is given by the expression **`(*(matrix+i)+j)`** which signifies the **address location** of the element **`matrix[i][j]`**. Next a nested for loop is used to traverse each element of the array to determine if it is unitary. If an element with i ≠ j (like matrix[1][2]) has a non-zero value, then a **`flag`** variable is set to 1, to indicate that the matrix is not a unitary matrix. Similarly if an element with i = j (like matrix[2][2]) is not equal to 1, then also the matrix is non-unitary and the **`flag`** is set to 1. Finally the matrix is displayed by another nested loop and by checking the value of the **`flag`** it is printed whether the matrix is unitary or not.

8.10 Pointers and Strings:

A string is basically a character array and hence it is natural that we can control strings efficiently using pointers as array operations can be done efficiently using pointers. We know that a constant string can be declared using the following array declaration:

```
char prompt[] = “Press any key to continue:”;
```

However instead of using a character array, we can use a character pointer to store the same string by using the following declaration:

```
char *prompt = “Press any key to continue:”;
```

The above declaration creates a pointer to a variable of type **`char`** called **`prompt`**, which is then initialised with the address of the constant array containing the characters **`“Press any key to continue:”`**.

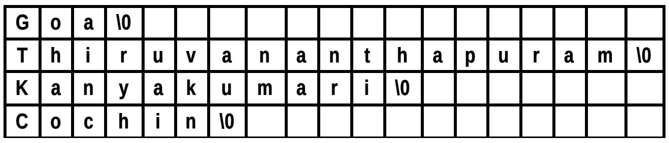
Using the above notation we had actually reduced a one-dimensional array to a single variable namely the pointer prompt. However in the above representation, in addition to the memory space used to store the constant array, extra memory is needed to store the pointer prompt, which will contain the address of the first element of the constant array. Representing a single constant string using a character pointer may not be of much importance. However, the main utility of this technique comes when we try to store multiple constant strings as shown below. Consider the following declaration:

```
char city[4][19] = {“Goa”, “Thiruvananthapuram”, “Kanyakumari”, “Cochin”};
```

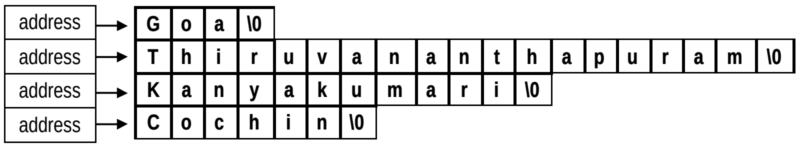
When a two dimensional character array like the above one is declared, the compiler reserves 4*19=76 bytes of memory to store the different names. It reserves 19 bytes for each name (including ‘\0’), though some names may be less than that (as compared to Thiruvananthapuram which contains 18 letters, the word Goa has only 3 letters). In doing so some memory is wasted as in evident from the diagram in the next page. This drawback can be eliminated if we use pointer notation to store the names as shown below:

```
char *city[4] = {“Goa”, “Thiruvananthapuram”, “Kanyakumari”, “Cochin”};
```

In the above technique, we define a **one-dimensional array of character pointers** to store the starting address locations of each of the strings. In doing so the strings are stored without any wasted space. The compiler reserves space for storing each string plus an array of four pointers. Whether using a pointer array instead of a character array saves memory will depend on the lengths of the string (Usually 4 bytes are used to store a pointer). In our example we have actually saved (19x4) – (4+19+12+7+4x4) = 76 – 58 =18 bytes. However the **problem** with pointer arrays is that the **strings cannot be changed later** in the program.



Accessing Strings using normal 2D character array



Accessing Strings using a 1D array of character pointers

As pointers provide an efficient and compact technique to access arrays, **programming operations with character strings are almost invariably done with pointers**. The following program simply copies the contents of one string called `input_string` into another string called `output_string`.

```
#include <stdio.h>
int main()
{ char input_string[80];
  char output_string[80];
  char *p_in = input_string;      → Pointer p_in initialised with address of input_string
  char *p_out = output_string;    → Pointer p_out initialised with address of output_string

  printf("\nEnter the string to copy");
  gets(input_string);

  while(*p_out++ = *p_in++);      → input_string copied to output_string using pointers

  puts(output_string);
  return 0;
}
```

The main part of the above program is the condition part of the **`while`** statement. The contents of the location pointed to by the pointer **`p_in`**, i.e. ***`p_in`** is copied to the contents of the memory location pointed to by the pointer **`p_out`** i.e. to **`*p_out`**, and the value checked. If the content of **`*p_out`** is non-zero (i.e. not **`NULL`**), then the condition evaluates to **true** and the pointers are incremented by the **`++`** operator to point to the next elements and the condition again checked. This continues till the **`NULL`** character **`\0`** is copied to **`*p_out`**, which evaluates to zero or **false** and terminates the loop.

8.11 Dynamic Memory Allocation:

When a variable is declared in a program then during runtime a memory space is reserved for storing the value assigned to the variable. However it may so happen that during writing the program it may not be certain as to how many variables will be required during runtime. For example a program may be used to calculate the standard deviation of a set of numbers. Each time the program is run, the user may have a different number of values. One way to overcome this problem is to declare an array containing a large index. But the problem with this approach is that when the program is run a fixed amount of memory location will always be reserved for use by the array. Suppose an integer array is declared with index 100. Then a memory space of (100x2) = 200 bytes will be reserved in the memory. In case the user inputs 10 values then the remaining (200-10x2) = 180 bytes will be simply wasted.

Under such circumstances the best solution would have been to be able to dynamically create variables i.e. to **create variables during run time as per requirement**. C provides us with some special memory allocating functions defined in the standard library stdio.h. The unused memory in the computer when a program is executed is usually called the **Heap**. When memory is required during runtime to store a variable, C allocates space for the variable from the heap. To do so C uses the following library functions:

`malloc()`: This function allocates a block of memory on the heap. The function takes an integer **`n`** as its argument and allocates a memory block equal to **`n`** bytes. The function returns a pointer indicating the address of the memory block. If the **allocation fails**, it returns a **`NULL`** value.

calloc(): This is very similar to **malloc()**. However it **takes two arguments**. The first argument indicates the total number of variables to be declared and the second argument indicates the size of each variable type. However unlike **malloc()**, after reserving the memory block, **calloc()** simultaneously **initialises the memory locations to zero**. If the **allocation fails**, it also returns a **NULL** pointer.

realloc(): This function is used to extend the size of a block of memory that has been previously declared. The first argument is a pointer to the block concerned and the second argument specifies the new size of the block in bytes.

free(): Once a program section has finished working with a block of memory, it needs to be freed, so that the same memory location can be used by other processes. The argument of the function is a pointer indicating the memory block that needs to be freed.

The following program piece uses the **calloc()** function to allocate a block of memory **to hold the marks of a variable number of students**. The number of students may change for each run of the program. Hence to optimise memory utilisation, the **calloc()** function is used to allocate the exact number of space to store the marks.

```
#include <stdio.h>

int main()
{ int *p_marks, num, sum=0, i;

  printf("\nEnter total number of students: ");
  scanf("%d", &num);

  p_marks = (int *)( calloc(num, sizeof(int)) );

  if(p_marks == NULL)

    {printf(“\nMemory allocation was not possible”);
     exit(0);
    }

  for(i=0; i<num; i++)
      {printf("Enter marks of student-%d: ", (i+1));
       scanf("%d", p_marks);
       sum = sum + *p_marks;
       p_marks++;
      }

  printf("\n\nAverage Marks = %f", (float)sum/num);

  free(p_marks);

  return 0;
}
```

The function **calloc()** is used to allocate memory to store **num** number of integers i.e. it allocates **numx2** bytes of memory space. The **sizeof()** function acting on the variable **int** gives the size of an integer i.e. 2 bytes. The function basically returns a **void pointer** i.e. a pointer which is not associated with any particular data type. To assign the pointer to point to integer type of variables, it is cast to an integer pointer using the casting **(int *)**. The final integer pointer is then assigned to the pointer variable **p_marks**. In case sufficient memory is not available, the function **calloc()** returns a **NULL** pointer. This condition is checked and if found true, the program is terminated by the **exit(0)** statement.

Next the marks are entered into the memory locations pointed to by **p_marks** through the **scanf()** function. Note that **p_marks** being a pointer which itself indicates an address location, no **&** is used before **p_marks** in **scanf()**.

The sum of the marks is next calculated and the pointer incremented to point to the next location by the **p_marks++** statement. In this way first all the marks are read and added to **sum**. Finally the average is calculated by the expression **(float)sum/num)**. Note that the integer variable **sum** is cast to a **float** to get the proper average. Otherwise the division of two integers would have given a wrong result.

Finally the reserved memory area is freed by using the **free()** function with **p_marks** as its argument.

The function **malloc()** can also be used to perform the above memory allocation. Then the memory allocating statement would be:

```
p_marks = (int *)( malloc(num*sizeof(int)) );
```