Pointers

Computer's Memory

A PC's RAM consists of many thousands of sequential storage locations, and each location is identified by a unique address. The memory addresses in a given computer range from 0 to a maximum value that depends on the amount of memory installed. When we're using our computer, the operating system uses some of the system's memory. When we're running a program, the program's code (the machine-language instructions for the program's various tasks) and data (the information the program is using) also use some of the system's memory. This section examines the memory storage for program data.

When we declare a variable in a C++ program, the compiler sets aside a memory location with a unique address to store that variable. The compiler associates that address with the variable's name. When our program uses the variable name, it automatically accesses the proper memory location. The location's address is used, but it is hidden from us, and we need not be concerned with it.

What Is a Pointer?

A Pointer is a variable which contains address of other variable.

Pointer Operators:

- The '&' or address operator, is a unary operator that returns the address of its operand.
- The '*' operator, commonly referred as indirection operator , returns the value of the object to which its operand (a pointer) points.

Rules Regarding pointer:

1. A pointer should be declared before its use like any other variable. A pointer is declared using '*' operator as follows:

```
type_name   *variable_name;
Eg. int x; // a integer variable
    int *y;  // a integer pointer variable
```

Here `x` is a variable declared as `int` and `y` is a pointer variable which can be assigned the address of variable declared as `int`. We can also say that `y` is a pointer to `int`.

2. A pointer can be declared as `char`,`int`, `short int`,`long int`, `float` etc. A pointer declared as one type can be assigned the address of variable declared as same type only. It is wrong to assign address of one type variable to other type pointer.
3. To assign the address of a variable to pointer, we use '&' operator .

Example:

```
int  a, *b;
float  c, *d;
double e, *f;
b = &a ;        // assigns the address of a to pointer b
d = &c ;       //assigns the address of c to ptr d
f = &e ;      //assigns the address of e to ptr f
```

4. The address operator '&' only applies to the variables and array elements. It cannot be applied to the expressions, constants or register variables.

5. To fetch the value of a variable using pointer , we are required to use the indirection operator '*'. Once pointer is assigned the address of a variable, we can fetch the variable by putting indirection operator before the pointer variable.

   Example:

   ```
   int a, *b;
   a = 10;
   b = &a;
   ```

   Now , `*b` points to `'a'` and can occur in any context where `'a'` can.
   For example: `*b =12;` results the value of `'a'` equal to `12;`

6. Since pointer are also variables they can also be used without indirection operator '*' to assign address from one pointer to another pointer variable.
   Example:

   ```
   int  a;
   int *b, *c;
   ```

   Declares one variable 'a' of type `int` and two pointer variables `'b'` and `'c'` to point `int` type variable.

   `a =10;` assigns the value 10 to variable 'a'.

   `b= &a;` assigns the address of 'a' to pointer 'b' i.e. b points to 'a'.

   Now, `c = b;` assigns the content of 'b' to 'c' i.e. address of 'a' to 'c'. So the pointers 'c' and 'b' both now points to a. and *b and *c both represent the value 10.

A simple Program:

```
 // Using the & and * operators
#include <iostream.h>

void main()
{
   int a;          //a is an integer
   int *aPtr;      // aPtr is a pointer to an integer
   a = 7;          // a is initialized to 7*/
   aPtr = &a;     // aPtr set to address of a */
   cout<<"The address of a is "<<&a<<"\nThe value of aPtr is "<<aPtr;
   cout<<"\n\nThe value of a is "<<a<<"\nThe value of *aPtr is "<< *aPtr;
   cout<<"\n\nShowing that * and & are inverses of each other."<<endl
   cout<<"&*aPtr ="<<&*aPtr<<"\n*&aPtr = "<<*&aPtr ;

 }
```

The output of the program looks like:

```
The address of a is 0x1b57248a
The value of aPtr is 0x1b57248a

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr =0x1b57248a
*&aPtr = 0x1b57248a
```

Another Example:

```
#include<iostream.h>
void main()
{
int a, *b;
a=5;
b=&a;
cout<<"The value of a is "<<a<<endl;
cout<<"The value of address of a is "<<&a<<endl;
cout<<"The value of *b is "<<*b<<endl;
cout<<"The value of address of variable pointed by b is "<<b<<endl;
*b+=3;
cout<<"The value of a = "<<a<<" *b = "<<*b<<endl;
a+=2;
cout<<"The  value of a= "<<a<<","<<"b = "<<*b;
}
```

The out put of this program is:

The value of a is 5
The value of address of a is 0x26772490
*The value of *b is 5*
The value of address of variable pointed by b is 0x26772490
*The value of a = 8 *b = 8*
The value of a= 10,b = 10

Algebra of Pointers:

RULE 1:

If '*'(indirection operator) and '&' (address operator) placed together , they cancel each other.

Example:

Let `m = &n;`

Then `*m = *&n`

As *& cancel each other, `*m = n.` So once we say `m =&n,` *m and n are identical and refer to the same variable

Arrays and Pointers:

The relationship between arrays and pointers is very strong. So , arrays and pointers can be used interchangeably in C/C++ and discussed simultaneously.

The array name always points to the first element of that array. For example `int a[10];` is the declaration of an integer array a of size 10.Then **`a`** points to **`a[0]`**; i.e. **a** and **&a[0]** refer same.

Here is an example to clear the concept of relation ship of arrays and pointer.

```
#include<iostream.h>
void main()
{
int a[5]={11,12,13,14,15};
int *b,i;

b=&a[0];
for(i=0;i<5;i++)
```

```
        cout<<a[i]<<" ";
cout<<endl;
for(i=0;i<5;i++)
        cout<<*(b+i)<<" ";
cout<<endl;
for(i=0;i<5;i++)
        cout<<*(a+i)<<" ";
cout<<endl;
for(i=0;i<5;i++)
        cout<<b[i]<<" ";
cout<<endl<<endl;
for(i=0;i<5;i++)
        cout<<(a+i)<<" ";
cout<<endl;
for(i=0;i<5;i++)
        cout<<(b+i)<<" ";
cout<<endl;
}
```

The output of this program:

11 12 13 14 15
11 12 13 14 15
11 12 13 14 15
11 12 13 14 15

0x227f23f6 0x227f23f8 0x227f23fa 0x227f23fc 0x227f23fe
0x227f23f6 0x227f23f8 0x227f23fa 0x227f23fc 0x227f23fe

Assigning address of array to pointer:

Algebra of pointers

RULE 2:

Let 'm' be declared as an array or as pointer then
&m[i] is equivalent to (m+i)
*m[i] is equivalent to *(m+i)*

In the above program,
`b=&a[0];` assigns the address of first element of array to the pointer 'b'.
also `b = a` ; is same as '`b= &a[0];`'
To fetch the value of first element we can write:

```
a[0]   or   *b
```

Similarly for next element we can write:

```
a[1]   or *(b+1)     and &a[1] is same as b+1
a[2]   or  *(b+2)    and &a[2] is same as b+2
```

In general, `a[i]` is same as `*(b+i)` , i= 1,2,3,4,…. And &a[i] is same as b+i

Array name can be used as pointer:

- We can use array name as pointer. In above program we can see `(a+i)` exactly does same task as `(b+i)` even if a is not declared as pointer. So we can use `*(a+i)` in place of `a[i]` since a is itself a pointer to that array a.
- Also pointer name can be used as array. In above program `b[i]` does exactly same as `a[i]` does even if b is not declared as array.

Are arrays and pointer same ?

Although, array name can be used as pointer and pointer name can be used as array. But arrays and pointers are not same they can be used interchangeably to represent same things. Followings are the differences between arrays and pointers:

1. When an array is declared like
   `int a[10];`
   Compiler allocates storage for 10 integers where as we declare i.e. 10*2 =20 bytes
   `int *b;`
   Storage is allocated for only one pointer i.e. only 4 bytes of memory. To allocate storage, we have to use **`new`** operator as,
   `b=new int[10];`
2. A pointer is a variable so we can write:
   `b = b+2;` etc
   where as array name is not a variable so it is not valid to write `a = a + 2;`

Multidimensional Array and Pointers:

Algebra of Pointers:

RULE #3

Let m is declared as array then

`m[i][j]` ➔ `*(*(m+i)+j); // two dimensional`
`m[i][j][k]` ➔ `*(*(*(m+i)+j)+k) // three dimensional`
`m[i][j][k][l]` ➔ `*(*(*(*(m+i)+j)+k)+l)` and so on..

We can access any element of a multidimensional array using pointer. Suppose the declaration of an array like:

```
int m[20][20];
```

Now , we want to fetch the element `m[10][8]`. Fetching this element from array using pointer can be done as:

Let us substitute `m[10]` with **`x`**.

`m[10][5]` ➔ `x[5]` on substituting `m[10]` with x
➔ `*(x+5)` as per the Rule 2 of pointer Algebra
➔ `*(m[10]+5)` on re-substituting x with `m[10]`.
➔ `*(*(m+10)+5)` on substituting m[10] with `*(m+10)` as per rule 2 of Ptr Algebra
Hence the value of m[10][5] can be fetched using array name as pointer by using
`*(*(m+10)+5)` which is equivalent to `m[10][5]`.

If we use array and pointer together , we can fetch the value of `m[10][5]` by using any of the following.

`m[10][5]` ➔ `*(m[10]+5)` or

`m[10][5]` ➔ `(*(m+10))[5]` or

`m[10][5]` ➔ `*(*(m+10)+5)`

In general `m[i][j]` ➔ `*(*(m+i)+j)`
Similarly in case of three dimensional array:
`m[i][j][k]` ➔ `*(*(*(m+i)+j)+k)`

Address Arithmetic

`If` p is pointer to some element of an array, then `p++` increments the pointer to point the next element, and `p+=i`, increments the p to point i elements beyond where it is currently pointing. These and similar arithmetic constructions performed with pointer is called pointer or address arithmetic.

Let us take example of a typical machine where the length of `char` is 1 byte, `int` is 2 byte, `long int` is 4 byte , `float` is 4 byte and `double` is 8 bytes.
Let us declare arrays as follows:

```
char ac[5];
int ai[5];
long int al[5];
float af[5];
double ad[5];
```

Now, if address of ac[0] is 5545, the address of ac[1] should be 5546, of ac[2] is 5547 etc . because the length of char is 1 bytes.

Similarly, if the address of `ai[0]` is say 6000, the address of `ai[1], ai[2]` should be 6002, 6004 and so on. Since the length of int is 2 bytes.

So the address of the next element of array depends upon the size. In address arithmetic,
`&ai[0]` ➔ `ai` and
`&ai[1]` ➔ `(ai+1)`
So the address of ai[0] is ai and ai[1] is (ai+1) irrespective of its type.

An Example:

```
//Program to illustrate address arithmetic
#include<iostream.h>
void main()
{
  char ac[5];
  int  ai[5],i;
  long int al[5];
  float af[5];
  double ad[5];

cout<<endl<<"The address of char array are :"<<endl;
for(i=0;i<5;i++)
     cout<<long(&ac[i])<<" ";
```

```
cout<<endl<<"The address of int array are "<<endl;
for(i=0;i<5;i++)
     cout<<long(&ai[i])<<" ";
cout<<endl<<"The address of long int  array are :"<<endl;
for(i=0;i<5;i++)
     cout<<long(&al[i])<<" ";
cout<<endl<<"The address of float  array are :"<<endl;
for(i=0;i<5;i++)
     cout<<long(&af[i])<<" ";
cout<<endl<<"The address of double array are :"<<endl;
for(i=0;i<5;i++)
     cout<<long(&ad[i])<<" ";
           }  //end of main()
```

The Output of the program looks like:

```
The address of char array are :
805250732 805250733 805250734 805250735 805250736
The address of int array are
805250722 805250724 805250726 805250728 805250730
The address of long int  array are :
805250662 805250666 805250670 805250674 805250678
The address of float  array are :
805250642 805250646 805250650 805250654 805250658
The address of double array are :
805250602 805250610 805250618 805250626 805250634
```

Observe the above output:

1. The address of char is increasing by 1
2. The address of int is increasing by 2
3. The address of long int is increasing by 4
4. The address of float is increasing by 4
5. The address of double is increasing by 8

So the address arithmetic is consistent and regular. Address is increasing depending upon the type of variable. When we write `ai+i`, i is scaled according to the size of the objects `ai` points to which is determined by the declaration of `ai`. If the length of `int` is 2 bytes, it will be scaled by 2.

So we can use
`(ai+i)` in place of `&ai[i]`
`(ac+i)` in place of `&ac[i]`
`(al+i)` in place of `&al[i]` and so on in above program.

Arrays of Pointer:

Pointer can be used in array. We can use arrays of pointer which is illustrated by the following example.

```
//This program illustrates the array of pointers
```

```
#include<iostream.h>
char name[][10]={"Ram","shyam","Hari", "Gopal", "Sita", "Gita"};
char *pname[10]= {"Ram","shyam","Hari", "Gopal", "Sita", "Gita"};

void main()
{
     int i;
     for(i=0;i<5;i++)
          cout<<&name[i][0]<<" ";
     cout<<endl<<"The names by pname:"<<endl;
     for(i=0;i<5;i++)
       cout<<&pname[i][0]<<" ";
}
```

The output:

```
Ram     shyam   Hari    Gopal   Sita
The names by pname:
Ram     shyam   Hari    Gopal   Sita
```

Refer the above program where in

`&name[i][0]`

`&pname[i][0]`

As per the rule 3 of pointer algebra,

`&name[i][0]` ➔ `&*(*(name+i)+0)` or `&*(name[i]+0)`

➔ `(*(name+i)+0)` or `(name[i]+0)` since &* cancel each other.

➔ `*(name+i)`

So we can write `&name[i][0]` as `*(name+i)`

Similarly, `&pname[i][0]` as `*(pname+i)`

So in above program we can simply write output statement as

```
Cout<<name[i]<<" ";  or cout<<*(name+i)<<" ";
 and
cout<<(pname+i)<<" ";  or cout<<*(pname+i)<<" ";
```

- The Above shows the similarities between the array of pointer to two dimensional array.

Difference between one dimensional pointer array and two dimensional array:

In the above example, in case of '**name**' which is declared as two dimensional array, storage is allocated for 5*10 = 50 characters.. Where as in case of '**pname**' which is declared as one dimensional pointer array, storage is allocated for 5 pointers. Because of initialization of pointer array at the time of declaration, storage is allocated as:

`pname:`

- ➔ `Ram\0`
- ➔ `shyam\0`
- ➔ `Hari\0`

➔ `Gopal\0`
➔ `Sita\0`

So, no space is wasted. But, in case of two dimensional array, space is allocated as follows:

➔`Ram\0……`
➔`shyam\0….`
➔`Hari\0…..`
➔`Gopal\0….`
➔`Sita\0…..` i.e. 10 chracter space is allocated for each name.

so space is wasted. In case of two dimensional array, space is reserved for 10 characters for each strings.

How to access the individual character in a string ?

Using Array index notation

- To access first character of first string use `pname[0][0]`
- To access second character of first string use `pname[0][1]`
- To access i^{th} character of first string use `pname[0][i]` and so on till `pname[0][i]` in not equal to '\0'

In general ,to access the j^{th} character of i^{th} string use `pname[i][j]`

Using pointer notation :

- To access the first character of first string, use `**pname`
- To access the second character of first string use `*(*pname+1)`
- To access the third character of first string use `*(*pname+2)`
- To access the i^{th} character of first string use `*(*pname+i)` and so on.

In general, to access the j^{th} character of i^{th} string use `*(*(pname+i)+j)`

Function Pointers

A function is not a variable but it is allowed to define pointer to a function like a variable. The function pointer is declared and used as mentioned below.

Let a function **`square(int)`** used to compute and return the square of an integer number defined as

```
int squqre(int n)
{
     return n*n;
}
```

the pointer to function is declared as,

`data type (*ptrname)(parameter type);`

e.g. `int (*fptr)(int);` Note that the parentheses around *fptr is essential. Without parentheses,
`int *fptr(int);` means: fptr is a function returning a pointer to integer which is different from pointer to function.

Assigning the address of function square() to function pointer *fptr is done as

`fptr =square;` // just name of function only. The function name itself represents the address of the function.

The function is invoked by using pointer to function as:
`Sq = (*fptr)(n);` where n is any integer value and sq is a integer variable. This statement is equivalent to
`Sq= square(n);`

Look at this example:

```
#include<iostream.h>
int square(int n)   // square function
{
     return n*n;
}

void main()
{
     int x;
     int (*fptr)(int); // fptr is pointer to a function.
     fptr=square;       // assigning address of square()
     cout<<"Enter integer x:";
     cin>>x;

     int sq;
     sq =(*fptr)(x);
     cout<<"The square of "<<x<<" is: "<<sq<<endl;
}
```

Output:

```
Enter Integer x:5
The square of 5 is: 25
```

The function pointer can be used as function arguments. Following example shows about this concept.
/* implementing the function pointer */

```
#include<iostream.h>
void main()
{
     int num1;
     char c;
```

```
     long int compute(long int (*)(int),int); //prototype
     long int square(int);
     long int cube(int);
     cout<<"Enter a number:";
     cin>>num1;
     cout<<"Compute square/Cube: input s or c : ";
     cin>>c;
     long int num2;
     if(c=='s'||c=='S')
          num2=compute(square,num1);  //passing address of
                                      //square function
     else
          num2 =compute(cube,num1); ////passing address of
                                      //cube function
     cout<<"The result ="<<num2<<endl;
}
long int compute(long int (*func)(int),int value)
{
     long int result;
     result =(*func)(value);
     return result;
}

long int square(int n)
{
     return n*n;
}

long int cube(int n)
{
     return n*n*n;
}
```

The output of program:
//On input character – C, output is

```
Enter a number:6
Compute square/Cube: input s or c : c
The result =216    */
```

//On input character – s, output is

```
Enter a number:8
Compute square/Cube: input s or c : s
The result =64
```