OBJECT ORIENTED PROGRAMMING LAB

Lab Manual # 05

Structures & Pointers in C++

Instructor: Hurmat Hidayat

Semester Spring, 2022

Course Code: CL1004

Fast National University of Computer and Emerging Sciences
Peshawar

Department of Computer Science

OBJECT ORIENTED PROGRAMMING LANGUAGE

Table of Contents

Structure

Structure is a collection of variables under a single name. Variables can be of any type: int, float, char etc. The main difference between structure and array is that arrays are collections of the same data type and structure is a collection of variables under a single name.

Why we need Structures?

As we noticed arrays are very powerful device that allow us to group large amounts of data together under a single variable name.

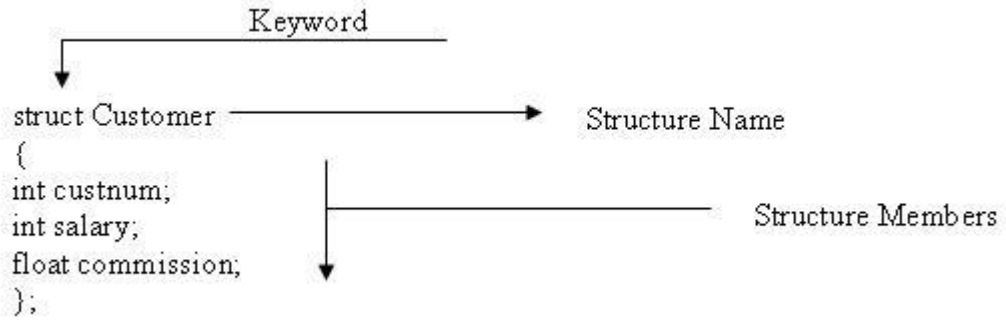
How would you write a program if, instead of being asked for simple list of either integers or characters, you were asked to combine integers, floating point numbers, and string with one variable name? You could not do it by using array.

It is actually quite often we want to group logically connected data that are of different types together. Just think about writing a program to store student record. You would need string for your name, integers to store the ID number, and a floating-point number to store the grades.

Declaring a Structure

Example:

Three variables: custnum of type int, salary of type int, commission of type float are structure members and the structure name is Customer. This structure is declared as follows:



In the above example, it is seen that variables of different types such as int and float are grouped in a single structure name Customer.

Arrays behave in the same way, declaring structures does not mean that memory is allocated. Structure declaration gives a skeleton or template for the structure.

```
struct st_name
{
        type l;
        type m;
        type n;
} ;
```

Where

st_name Represents the structure name. It is also called **Structure tag**. It is used to declare variables of structure type.

type Represent the type of the variable.

l,m,n Represent members of the structure. These may have different or same data type. The members are enclosed in braces.

A semicolon after the closing bracket (**} ;**) indicates the end of the structure.

Each member of a structure must have unique name but different structures may have same names.

A structure is first defined and then variables of structure type are declared. A variable of a structure type is declared to access data in the members of the structure.

Example: Declare a structure with ***address*** as a tag and having two members ***name*** of character type and ***age*** of integer type.

```
struct address
{
        char name [15];
        int age;
} ;
```

The structure member **name** is of character type with 15 characters length (including the null character) and **age** is of integer type.

Structure Variables

After declaring the structure, the next step is to define a structure variable.

A structure is a collection of data items or elements. It is defined to declare its variables. These are called **structure variables**. A variable of structure type represents the members of its structure.

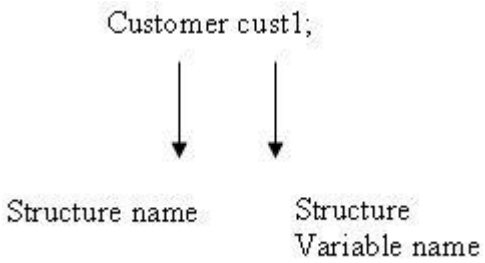
When a structure type variable is declared a space is reserved in the computer memory to hold all members of the structure. The memory occupied by a structure variable is equal to the sum of the memory occupied be each member of the structure.

A structure can be defined prior to or inside the main() function. If it is defined inside the main() function then the structure variable can be declared only inside the main function. If it is declared prior to the main() function, its variables can be defined anywhere in the program.

```
struct address

{

        char city[15];

        int pcode;

} ;

address taq , aye;
```

In the above example, **city** and **pcode** are members of the structure **address**. The variable **taq** and **aye** are declared as structure variables.

The structure **address** has two members **city** and **pcode**. The variable **city** occupies 15 bytes and **pcode** occupies 2 bytes. Thus, each variable of this structure will occupy 17 bytes space in memory, i.e. 15 bytes for **city** and 2 bytes **pcode**.



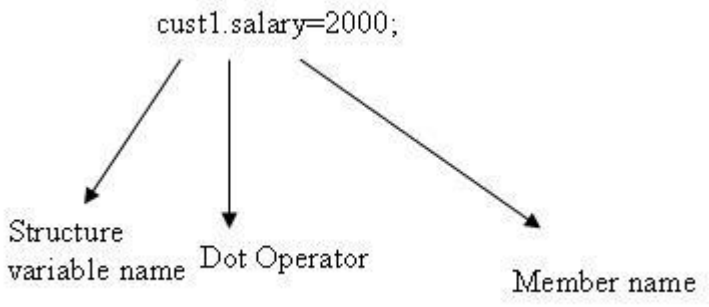
How to access structure members in C++?

To access structure members, the operator used is the dot operator denoted by (.). The dot operator for accessing structure members is used thusly:

```
Structure_Variable_name.member_name;
```

For example:

A programmer wants to assign 2000 for the structure member salary in the above example of structure Customer with structure variable cust1 this is written as:




```
// A Complete example by declaring, using structs as well as data members.
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
struct Employee
{
// members of the structures
string name;
double hours_worked;
double salary;
};
int main()
{
      Employee a,b; //structure variables

      //accessing structure variables
      a.name="Asia";
      a.hours_worked=30;
      a.salary=a.hours_worked*50;

      b.name="Naveed";
      b.hours_worked=60;
      b.salary=b.hours_worked*50;
```

```
	//*for 1st employee
	cout<<"Employee Name:\t"<<a.name<<endl;
	cout<<"Hours worked:\t"<<a.hours_worked<<endl;
	cout<<"Salary:\t\t"<<a.salary<<endl;

	//*for second employee
	cout<<"Employee Name:\t"<<b.name<<endl;
	cout<<"Hours worked:\t"<<b.hours_worked<<endl;
	cout<<"Salary:\t\t"<<b.salary<<endl;

}

/*Sample Output

Employee Name:  Asia
Hours worked:   30
Salary:         1500
Employee Name:  Naveed
Hours worked:   60
Salary:         3000
```

Members of Structures

Structures in C++ can contain two types of members:

Data Member: These members are normal C++ variables. We can create a structure with variables of different data types in C++.

Member Functions: These members are normal C++ functions. Along with variables, we can also include functions inside a structure declaration.

Example:

```
// Data Members
int roll;
int age;
int marks;

// Member Functions
void printDetails()
{
	cout<<"Roll = "<<roll<<"\n";
	cout<<"Age = "<<age<<"\n";
	cout<<"Marks = "<<marks;
}
```

In the above structure, the data members are three integer variables to store *roll number, age and marks* of any student and the member function is *printDetails()* which is printing all of the above details of any student.

Function inside Structure

```
// Function inside structs
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
struct Employee
{
string name;
double hours_worked;
double salary;

//this function will initialize members of structure
void input(string n, double h)
{
      name=n;
      hours_worked=h;
}

//this function will print members of structure
void print()
{
cout<<"Employee Name:\t"<<name<<endl;
cout<<"Hours worked:\t"<<hours_worked<<endl;
salary=50*hours_worked;
cout<<"Salary:\t\t"<<salary<<endl;
}
};   //structure body closed
int main()
{
      Employee a;      //structure variable
      a.input("Aisha",10);
      a.print();
}
/*
Sample output:
Employee Name:  Aisha
Hours worked:   10
Salary:         500
*/
```

Structs of Arrays (Array within Structure)

```
#include<iostream>
#include<string>

using namespace std;

struct student {

      string name;
      int ID;
      int testScore[3]; // array within structure as member of struct

};


int main () {

//    declare structure variable of type student

      student st1;

// access and initialise st1
      st1.name = "Ahmad";

      st1.ID = 1010;

      st1.testScore[0] = 56;
      st1.testScore[1] = 45;
      st1.testScore[2] = 33;

// Display content of st1

      cout<< "\n Name: "<< st1.name <<endl;
      cout<< "\n ID: "<< st1.ID <<endl;

      cout<< "\n Score 1: "<< st1.testScore[0] <<endl;
      cout<< "\n Score 2: "<< st1.testScore[1] <<endl;
      cout<< "\n Score 3: "<< st1.testScore[2] <<endl;

      return 0;
}
```

```
Name: Ahmad
ID: 1010
Score 1: 56
Score 2: 45
Score 3: 33
```

Arrays of Structs

What is an array of structures?

Like other primitive data types, we can create an array of structures.

Instead of declaring multiple variables we can also declare an array of structure in which each element of the array will represent a structure variable.

```
#include <iostream>
#include <iostream>
#include<string>
using namespace std;

struct point {

	int x;
	int y;

};

int main () {

//	array of size 3, each element of the array is actually a structure of type
point

	struct point p[3];

//	consider a triangle with 3 points
	p[0].x = 5;
	p[0].y = 6;

	p[1].x = 8;
	p[1].y = 9;

	p[2].x = 11;
	p[2].y = 12;

//display content

	for (int i = 0; i < 3; i++){
```

```
            cout << "Point "<< i+1 << " (x,y) : (" << p[i].x << "," << p[i].y <<")"
<<endl;

      }

}

Output
Point 1 (x,y) : (5,6)
Point 2 (x,y) : (8,9)
Point 3 (x,y) : (11,12)
```

What is a structure pointer?

Like primitive types, we can have pointer to a structure. If we have a pointer to structure, members are accessed using arrow (->) operator instead of the dot (.) operator.

```
#include <iostream>
using namespace std;

struct Point {
    int x, y;
};

int main()
{
    Point p1, *p2;    // structure variables

    p1.x=2;
    p1.y=3;

    // p2 is a pointer to structure p1
    p2 = &p1;

    // Accessing structure members using
    // structure pointer
    cout << p2->x << " " << p2->y;
    return 0;
}

Output
2   3
```

Nested Structure in C++

When a structure contains another structure, it is called nested structure. For example, we have two structures named Address and Employee. To make Address nested to Employee, we have to

define Address structure before and outside Employee structure and create an object of Address structure inside Employee structure.

Syntax for structure within structure or nested structure

```
struct structure1
       {
            - - - - - - - - - -
            - - - - - - - - - -
       };
struct structure2
        {
             - - - - - - - - - -
             - - - - - - - - - -
             struct structure1 obj;
         };
```

Nested Structs Example 1

```
#include <iostream>
using namespace std;
struct Address
        {
               char HouseNo[25];
               char City[25];
               char PinCode[25];
        };

struct Employee
       {
           int Id;
           char Name[25];
           float Salary;
           struct Address Add;
       };

int main()
    {
        int i;
        Employee E;

        cout << "\n\tEnter Employee Id : ";
        cin >> E.Id;
```

```
        cout << "\n\tEnter Employee Name : ";
        cin >> E.Name;

        cout << "\n\tEnter Employee Salary : ";
        cin >> E.Salary;

        cout << "\n\tEnter Employee House No : ";
        cin >> E.Add.HouseNo;

        cout << "\n\tEnter Employee City : ";
        cin >> E.Add.City;

        cout << "\n\tEnter Employee House No : ";
        cin >> E.Add.PinCode;

        cout << "\nDetails of Employees";
        cout << "\n\tEmployee Id : " << E.Id;
        cout << "\n\tEmployee Name : " << E.Name;
        cout << "\n\tEmployee Salary : " << E.Salary;
        cout << "\n\tEmployee House No : " << E.Add.HouseNo;
        cout << "\n\tEmployee City : " << E.Add.City;
        cout << "\n\tEmployee House No : " << E.Add.PinCode;
    }
Output :
             Enter Employee Id : 101
             Enter Employee Name : Suresh
             Enter Employee Salary : 45000
             Enter Employee House No : 4598/D
             Enter Employee City : Delhi
             Enter Employee Pin Code : 110056
Details of Employees
             Employee Id : 101
             Employee Name : Suresh
             Employee Salary : 45000
             Employee House No : 4598/D
             Employee City : Delhi
             Employee Pin Code : 110056
```

Nested Structs Example 2

```
#include <iostream>
#include <conio.h>
using namespace std;
struct Distance
{
```

```
int feet;
float inches;
};
struct Room
{
Distance length;
Distance width;
};
void main()
{
Room dining; //define a room
dining.length.feet = 13; //assign values to room
dining.length.inches = 6.5;
dining.width.feet = 10;
dining.width.inches = 0.0;
//Alternative: Room dining = { {13, 6.5}, {10, 0.0} };
//convert length & width
float l = dining.length.feet + dining.length.inches/12;
float w = dining.width.feet + dining.width.inches/12;
//find area and display it
cout << "Dining room area is " << l * w<< " square feet\n" ;
system("pause");
}
/*Sample Run:
Dining room area is 135.417 square feet
*/
```

Structure and Function in C++

Using function, we can pass structure as function argument and we can also return structure from function.

Structure can be passed to function through its object therefore passing structure to function or passing structure object to function is same thing because structure object represents the structure. Like normal variable, structure variable (structure object) can be passed by value or by references / addresses

Passing Structure by Value

In this approach, the structure object is passed as function argument to the definition of function, here object is representing the members of structure with their values.

Example for passing structure object by value

```
#include<iostream>
using namespace std;
struct Employee
{
    int Id;
    char Name[25];
    int Age;
    long Salary;
};

void Display(Employee);   // function declaration
int main()
{
     // Initializing structure variables
     Employee Emp = {1,"Aisha",29,45000};
     Display(Emp);   // calling function
}

void Display(Employee E)
{
    cout << "\n\nEmployee Id : " << E.Id;
    cout << "\nEmployee Name : " << E.Name;
    cout << "\nEmployee Age : " << E.Age;
    cout << "\nEmployee Salary : " << E.Salary;
}
```

Output:

```
Employee Id : 1
Employee Name : Aisha
Employee Age : 29
Employee Salary : 45000
```

Function Returning Structure

Structure is user-defined data type, like built-in data types structure can be returned from function.

Example for Function Returning Structure

```
#include<iostream>
using namespace std;

struct Employee
{
         int Id;
         char Name[25];
```

```
        int Age;
        long Salary;
};

Employee Input();            //Statement 1
int main()
{
    Employee Emp;

    Emp = Input();  // calling function

    cout << "\n\nEmployee Id : " << Emp.Id;
    cout << "\nEmployee Name : " << Emp.Name;
    cout << "\nEmployee Age : " << Emp.Age;
    cout << "\nEmployee Salary : " << Emp.Salary;

}
Employee Input()
{
    Employee E;   // declaring structure variable

    cout << "\nEnter Employee Id : ";
    cin >> E.Id;

    cout << "\nEnter Employee Name : ";
    cin >> E.Name;

    cout << "\nEnter Employee Age : ";
    cin >> E.Age;

    cout << "\nEnter Employee Salary : ";
    cin >> E.Salary;

    return E;             //Statement   2
}
```

Output:

```
Enter Employee Id : 1
Enter Employee Name : Amir
Enter Employee Age : 23
Enter Employee Salary : 234235

Employee Id : 1
Employee Name : Amir
Employee Age : 23
Employee Salary : 234235
```

In the above example, statement 1 is declaring Input() with return type Employee. As we know structure is user-defined data type and structure name acts as our new user-defined data type, therefore we use structure name as function return type.

Input() have local variable E of Employee type. After getting values from user statement 2 returns E to the calling function and display the values.

Pointers

Pointers are the most powerful feature of C and C++. These are used to create and manipulate data structures such as linked lists, queues, stacks, trees etc. The virtual functions also require the use of pointers. These are used in advanced programming techniques. To understand the use of pointers, the knowledge of memory locations, memory addresses and storage of variables in memory is required.

Memory addresses & Variables

Computer memory is divided into various locations. Each location consists of 1 byte. Each byte has a unique address.

When a program is executed, it is loaded into the memory from the disk. It occupies a certain range of these memory locations. Similarly, each variable defined in the program occupies certain memory locations. For example, an int type variable occupies two bytes and float type variable occupies four bytes.

When a variable is created in the memory, three properties are associated with it. These are:

- Type of the variable
- Name of the variable
- Memory address assigned to the variable.

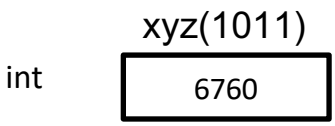
For example an integer type variable xyz is declared as shown below

int xyz =6760;

int represents the data type of the variable.

xyz represents the name of the variable.

When variable is declared, a memory location is assigned to it. Suppose, the memory address assigned to the above variable xyz is 1011. The attribute or properties of this variable can be shown as below:



The box indicates the storage location in the memory for the variable xyz. The value of the variable is accessed by referencing its name. Thus, to print the contents of variable xyz on the computer screen, the statement is written as:

```
cout<<xyz
```

The memory address where the contents of a specific variable are stored can also be accessed. The **address operator (&)** is used with the variable name to access its memory address. The address operator (&) is used before the variable name.

For example, to print the memory address of the variable xyz, the statement is written as:

```
cout<<&xyz
```

The memory address is printed in hexadecimal format.

Pointer Variables

The variables that is used to hold the memory address of another variable is called a pointer variable or simply pointer.

The data type of the variable (whose address a pointer is to hold) and the pointer variable must be the same. A pointer variable is declared by placing an asterisk (*) after data type or before the variable name in the data type statement.

For example, if a pointer variable **"p"** is to hold memory address of an integer variable, it is declared as:

```
int* p;
```

Similarly, if a pointer variable **"rep"** is to hold memory address of a floating-point variable, it is declared as:

```
float* rep;
```

The above statements indicate that both "**p"** and **"rep"** variable are pointer variables and they can hold memory address of integer and floating-point variable respectively.

Although the asterisk is written after the data type, is usually more convenient to place the asterisk before the pointer variable. i.e. **float *rep;**

```
#include <iostream>

using namespace std;
int main() {


int a, b;

int *x, *y;
a =  33;

b =  66;

x = &a;

y = &b;


cout<<"Memory address of variable a= "<<x<<endl;

cout<<"Memory address of variable b= "<<y<<endl;
  return 0;

}
```

Output:

```
Memory address of variable a= 0x7bfe0c

Memory address of variable b= 0x7bfe08
```

A pointer variable can also be used to access data of memory location to which it points.

In the above program, **x** and **y** are two pointer variables. They hold memory addresses of variables **a** and **b**. To access the contents of the memory addresses of a pointer variable, an asterisk (*) is used before the pointer variable.

For example, to access the contents of **a** and **b** through pointer variable **x** and **y**, an asterisk is used before the pointer variable. For example,

```
cout<<"Value in memory address x = "<<*x<<endl;

cout<<"Value in memory address y = "<<*y<<endl;
```

Program

Write a program to assign a value to a variable using its pointer variable. Print out the value using the variable name and also print out the memory address of the variable using pointer variable.

```
#include <iostream>

using namespace std;

int main () {

int *p;

int a;

p=&a;
cout<<"Enter data value? ";

cin>>*p;

cout<<"Value of variable    ="<<a<<endl;

cout<<"Memory Address of variable= "<<p<<endl;

   return 0;

}
```

Output:

```
Enter data value? 44

Value of variable    =44

Memory Address of variable= 0x7bfe14
```

The "void" Type Pointers

Usually type of variable and type of pointer variable that holds memory address of the variable must be the same. But the "**void**" type pointer variable can hold memory address of variables of any type. A void type pointer is declared by using keyword "**void**". The asterisk is used before pointer variable name.

Syntax for declaring void type pointer variable is:

void *p;

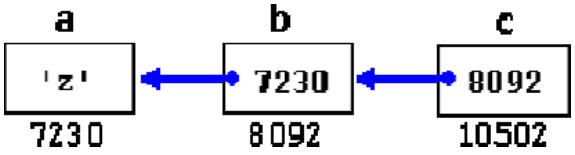
The pointer variable "**p**" can hold the memory address of variables of any data type.

Pointers to Pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```
char a;

void *b;

void  **
c;

a = 'z';

b = &a;

c = &b;
```

This, supposing the randomly chosen memory locations for each variable of 7230, 8092 and 10502, could be represented as:



The value of each variable is written inside each cell; under the cells are their respective addresses in memory. The new thing in this example is variable c, which can be used in three different levels of indirection, each one of them would correspond to a different value:

```
#include <iostream>

using namespace std;
int main ()
```

```
{
int a;
int *b;
int **c;
int ***d;
a = 7;
b = &a;
c = &b;
d = &c;

cout<<"The Address of the Vairiable a is: "<<b<<endl;
cout<<"The Address of the Vairiable b is: "<<c<<endl;
cout<<"The Address of the Vairiable d is: "<<d<<endl;
return 0;
}
```

Output:

```
The Address of the Vairiable a is: 0x7bfe14
The Address of the Vairiable b is: 0x7bfe08
The Address of the Vairiable d is: 0x7bfe00
```

The Reference (Address Of) Operator (&)

As soon as we declare a variable, the amount of memory needed is assigned for it at a specific location in memory (its memory address). We generally do not actively decide the exact location of the variable within the panel of cells that we have imagined the memory to be - Fortunately, that is a task automatically performed by the operating system during runtime. However, in some cases we may be interested in knowing the address where our variable is being stored during runtime in order to operate with relative positions to it.

The address that locates a variable within memory is what we call a reference to that variable. This reference to a variable can be obtained by preceding the identifier of a

variable with an ampersand sign (&), known as reference operator, and which can be literally translated as "address of". For example:

```
ted = &andy;
```

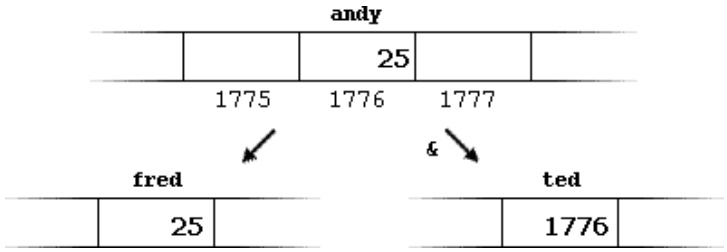
This would assign to ted the address of variable **andy**, since when preceding the name of the variable **andy** with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

From now on we are going to assume that andy is placed during runtime in the memory address 1776. This number (1776) is just an arbitrary assumption we are inventing right now in order to help clarify some concepts in this tutorial, but in reality, we cannot know before runtime the real value the address of a variable will have in memory.

Consider the following code fragment:

```
andy = 25;
fred = andy;
ted = &andy;
```

The values contained in each variable after the execution of this, are shown in the following diagram:



First, we have assigned the value 25 to andy (a variable whose address in memory we have assumed to be 1776). The second statement copied to fred the content of variable andy (which is 25). This is a standard assignment operation, as we have done so many times before.

Finally, the third statement copies to ted not the value contained in andy but a reference to it (i.e., its address, which we have assumed to be 1776). The reason is that in this third assignment operation we have preceded the identifier andy with the reference operator (&), so we were no longer referring to the value of andy but to its

reference (its address in memory).

The variable that stores the reference to another variable (like ted in the previous example) is what we call a pointer. Pointers are a very powerful feature of the C++ language that has many uses in advanced programming. Farther ahead, we will see how this type of variable is used and declared.

The Dereferencing Operator (*)

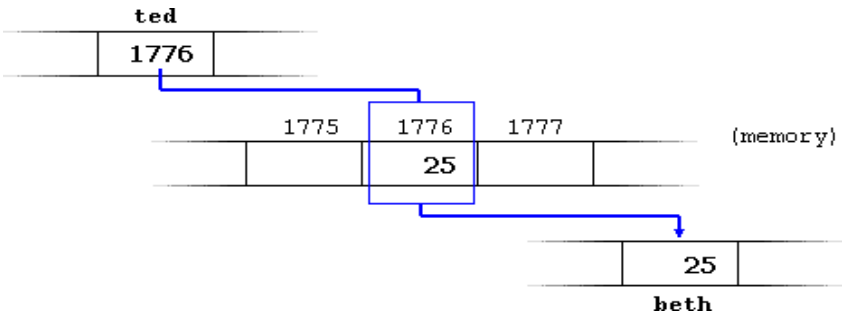
We have just seen that a variable which stores a reference to another variable is called a pointer. Pointers are said to "point to" the variable whose reference they store.

Using a pointer, we can directly access the value stored in the variable which it points to. To do this, we simply have to precede the pointer's identifier with an asterisk (*), which acts as dereference operator and that can be literally translated to "value pointed by".

Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") beth would take the value 25, since ted is 1776, and the value pointed by 1776 is 25.



You must clearly differentiate that the expression ted refers to the value 1776, while *ted (with an asterisk * preceding the identifier) refers to the value stored at address 1776, which in this case is 25. Notice the difference of including or not including the dereference operator (I have included an explanatory commentary of how each of these two expressions could be read):

```
beth = ted; // beth equal to ted ( 1776 )
```

Notice the difference between the **reference** and **dereference** operators:

- & is the **reference operator** and can be read as "**address of**"
- * is the **dereference operator** and can be read as "**value pointed by**"

Thus, they have complementary (or opposite) meanings. A variable referenced with & can be dereferenced with *.

Earlier we performed the following two assignment operations:

```
andy = 25;
ted = &andy;
```

Right after these two statements, all of the following expressions would give true as result:

```
andy == 25

&andy == 1776

ted  ==  1776
```

The first expression is quite clear considering that the assignment operation performed on andy was andy=25. The second one uses the reference operator (&), which returns the address of variable andy, which we assumed it to have a value of 1776. The third one is somewhat obvious since the second expression was true and the assignment operation performed on ted was ted=&andy. The fourth expression uses the dereference operator (*) that, as we have just seen, can be read as "value pointed by", and the value pointed by ted is indeed 25.
So, after all that, you may also infer that for as long as the address pointed by ted remains unchanged the following expression will also be true:

```
*ted  ==  andy
```

Pointers and Arrays

There is a close relationship between pointers and arrays. In Advanced programming, arrays are accessed using pointers.

Arrays consist of consecutive locations in the computer memory. To access an array, the memory location of the first element of the array is accessed using the pointer variable. The pointer is then incremented to access other elements of the array. The pointer is increased in the value according to the size of the elements of the array.

When an array is declared, the array name points to the starting address of the array. For example, consider the following example.

int x[5];

int *p;

The array" **x**" is of type **int** and "**p**" is a pointer variable of type **int**.

To store the starting address of array "x" (or the address of first element), the following statement is used.

p = x;

The address operator (&) is not used when only the array name is used. If an element of the array is used, the & operator is used. For example, if memory address of first element of the array is to be assigned to a pointer, the statement is written as:

p = &x[0];

when integer value 1 is added to or subtracted from the pointer variable "**p**", the content of pointer variable "p" is incremented or decremented by (1 x size of the object or element), it is incremented by 1 and multiplied with the size of the object or element to which the pointer refers.

For example, the memory size of various data types is shown below:

- The array of **int** type has its object or element size of **2 bytes**. It is **4 bytes** in Xenix System.
- The array of type **float** has its object or element size of **4 bytes**.
- The array of type **double** has its object or element size of **8 bytes**.
- The array of type **char** has its object or element size of **1 byte**.

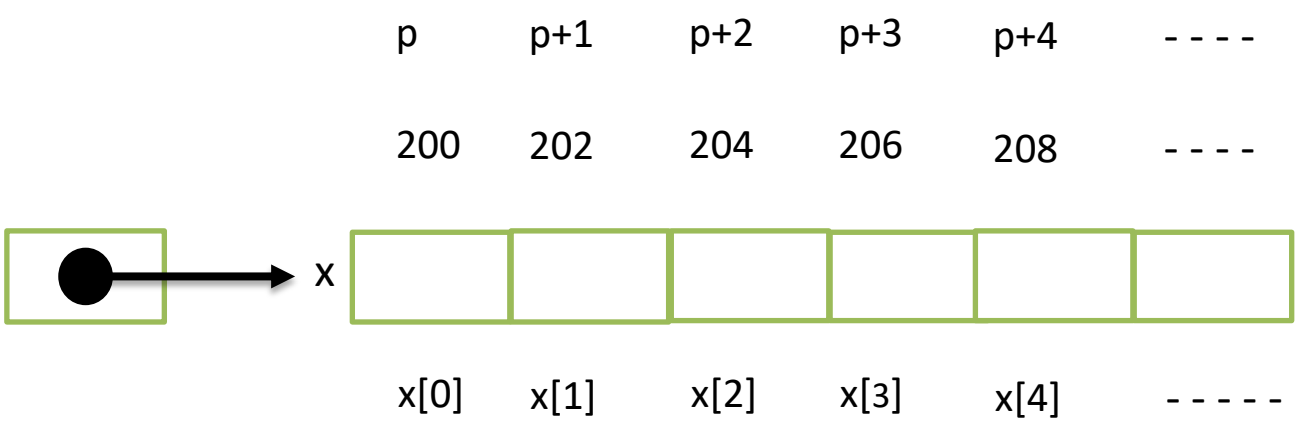
Suppose the location of first element in memory is 200. i.e., the value of pointer variable "p" is 200, and it refers to an integer variable.

When the following statement is executed,

p=p+1;

the newly value of "**p**" will be 200+(1*2)=202. All elements of an array can be accessed by using this technique.

The logical diagram of an integer type array "**x**" and pointer variable "**p**" is that refers to the elements of the array "x" is given below:



Program

Write a Program to input data into an array and then to print on the computer screen by using pointer notation.

```
#include <iostream>

using namespace std;

int main () {
int arr[5], *pp, i;

pp=arr;

cout<<"Enter Values into an array: "<<endl;
for(int i=0 ; i<=4 ; i++)

{

    cin>>arr[i];

}
cout<<"Values from array Using Pointer notation: "<<endl;
for(int i=0 ; i<=4 ; i++)

{

    cout<<*pp++<<"\t";
```

```
}
   return 0;

}
```

Output:

```
Enter Values into an array:

4

5

3

55

88

Values from array Using Pointer notation:

4       5       3       55      88
```

Passing Pointers as Arguments to Functions

The pointer variables can also be passed to functions as arguments. When pointer variable is passed to a function, the address of the variable is passed to the function. Thus, a variable is passed to a function not by its value but by its reference.

```
#include <iostream>

using namespace std;
void temp(int *, int *);

int main () {
int a, b;

a=10;

b=20;
temp(&a, &b);  // function calling
cout<<"Value of a= "<<a<<endl;

cout<<"Value of b= "<<b<<endl;

cout<<"OK"<<endl;
return 0;

}
void temp(int *x, int *y)

{
```

```
    *x = *x+100;
    *y = *y+100;
}
```

Output:

```
Value of a= 110
Value of b= 120
OK
```

Program Explanation In the above program, the function "**temp**" has two parameters which are pointers and are of **int** type. When the function **"temp**" is called, the addresses of variables "**a**" and "**b**" are passed to the function. In the function, a value 100 is added to both variables "**a"** and "**b**" through their pointers. That is, the previous values of variables "**a**" and "**b**" are increased by 100. When the control returns to the program, the values of variable **a** is 110 and that of variable **b** is 120.

Passing Pointers to a Function

```
#include <string>
#include <iostream>
using namespace std;
void abc(int *a)
{
*a=*a * *a - *a;
}
int main()
{
int x=5;
int *p;
p=&x;
abc(p); // calling function
cout<<"Value of p is changed by the function passed as parameter.: "<<*p<<endl;
}
```

Output:

```
Value of p is changed by the function passed as parameter.: 20
```

Program

Write a program to swap two values by passing pointers as arguments to the function.

```
#include <iostream>

using namespace std;
void swap(int*, int*); // Function prototype

int main () {
int a, b;
cout<<"Enter 1st Value for a ? ";

cin>>a;
cout<<"Enter 2nd Value for b? ";

cin>>b;
swap(&a, &b);  // Function Calling
cout<<"Values after exchange = "<<endl;

cout<<"Value of a= "<<a<<endl;

cout<<"Value of b= "<<b<<endl;
return 0;

}
void swap(int *x, int *y)   // Function Definition

{

    int t;

    t = *x;

    *x = *y;

    *y = t;

}
```

Output:

```
Enter 1st Value for a? 3

Enter 2nd Value for b? 7

Values after exchange =

Value of a= 7

Value of b= 3
```

Returning Pointers from Function

```
using namespace std;

#include <string>

#include <iostream>

using namespace std;

int* abc(int &a)

{

int *p;

p=&a;

*p = (*p + *p) * *p - *p**p;

return p;

}

int main()

{

int x=3;

int *p;

p=abc(x);

cout<<"Value of p is changed by the function returned.: "<<*p<<endl;

}
```

Output:

```
Value of p is changed by the function returned.: 9
```