Data Structures-LAB

Lab Manual # 01

Pointers in C++

Instructor: Engr. Khuram Shahzad

Semester Spring, 2022

Course Code: CL2001

Fast National University of Computer and Emerging Sciences Peshawar

Department of Computer Science

Data Structures-LAB

Table of Contents

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.

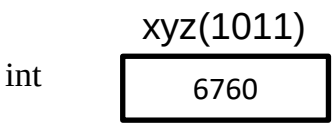
For example, an integer type variable xyz is declared as shown below

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;
```

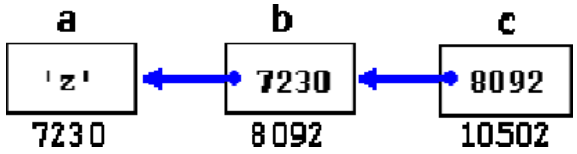
The pointer variable "**p**" can hold the memory address of variables of any data type.

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;

char *b;

char **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;
```

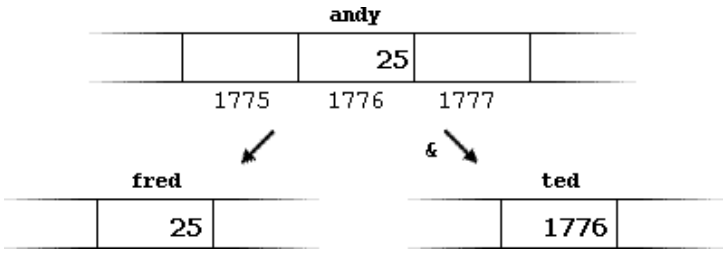
This would assign to ted the address of variable **andy**, since when preceding the name of the variable **andy** with the reference operator (&) we are no longer talking about the content of the variable itself, but about its reference (i.e., its address in memory).

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.

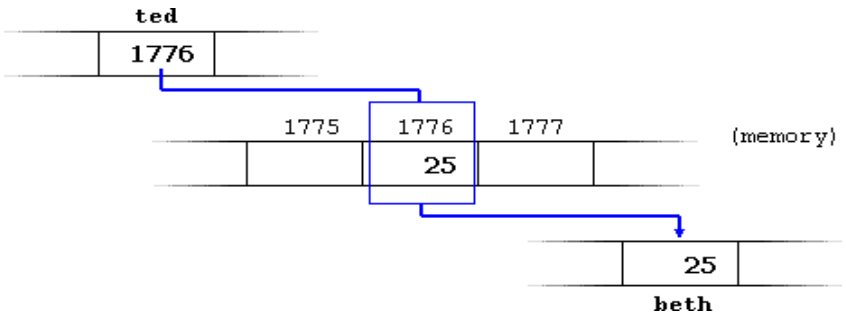
The Dereferencing Operator (*)

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**.

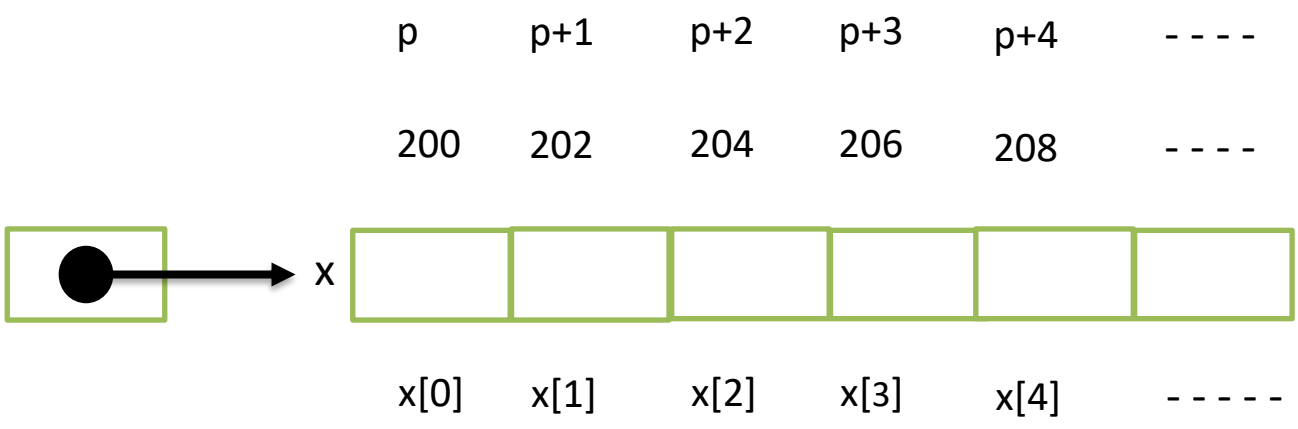
Suppose the location of first element in memory is 200. i.e., the value of pointer variable "p" is 200, and it refers to an integer variable.

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
```

String in C++

String is a collection of characters. There are two types of strings commonly used in C++ programming language:

- Strings that are objects of string class (The Standard C++ Library string class).
- C-strings (C-style Strings).

C-strings

In C programming, the collection of characters is stored in the form of arrays. This is also supported in C++ programming. Hence, it's called C-strings.

C-strings are arrays of type `char` terminated with null character, that is, `\0` (ASCII value of null character is 0).

How to define a C-string?

char str[ ] = "C++";

In the above code, `str` is a string and it holds 4 characters.

Although, "`C++`" has 3 character, the null character `\0` is added to the end of the string automatically.

cout<<str; // will print C++

cout<<str[0]; // will print only "**C**"

Alternative ways of defining a string

char str[4] = "C++";

char str[] = {'C','+','+','\0'};

char str[4] = {'C','+','+','\0'};

Like arrays, it is not necessary to use all the space allocated for the string. For example:

char str[100] = "C++";

Example 1: C++ String to read a word

C++ program to display a string entered by user.

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

int main()
{
   char str[100];

   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;

   cout << "\nEnter another string: ";
   cin >> str;
```

```
    cout << "You entered: "<<str<<endl;

    return 0;
}
```

Output

```
Enter a string: C++
You entered: C++

Enter another string: Programming is fun.
You entered: Programming
```

Notice that, in the second example only "Programming" is displayed instead of "Programming is fun".

This is because the extraction operator >> works as `scanf()` in C and considers a space " " has a **terminating character**.

Example 2: C++ String to read a line of text

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

int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);

    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

To read the text containing blank space, `cin.get` function can be used. This function takes two arguments.

First argument is the name of the string (address of first element of string) and second argument is the maximum size of the array.

In the above program, `str` is the name of the string and `100` is the maximum size of the array.

string Object

In C++, you can also create a string object for holding strings.

Unlike using char arrays, string objects has no fixed length, and can be extended as per your requirement.

Example 3: C++ string using string data type

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

int main()
{
   // Declaring a string object
   string str;
   cout << "Enter a string: ";
   getline(cin, str);

   cout << "You entered: " << str << endl;
   return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

In this program, a string `str` is declared. Then the string is asked from the user.

Instead of using `cin>>` or `cin.get()` function, you can get the entered line of text using `getline()`.

`getline()` function takes the input stream as the first parameter which is `cin` and `str` as the location of the line to be stored.

Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

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

void display(char *);
void display(string);

int main()
{
   string str1;
   char str[100];

   cout << "Enter a string: ";
   getline(cin, str1);

   cout << "Enter another string: ";
   cin.get(str, 100, '\n');

   display(str1);
   display(str);
   return 0;
}

void display(char s[])
{
   cout << "Entered char array is: " << s << endl;
}
```

```
void display(string s)
{
   cout << "Entered string is: " << s << endl;
}
```

Output

```
Enter a string:  Programming is fun.
Enter another string:  Really?
Entered string is: Programming is fun.
Entered char array is: Really?
```

In the above program, two strings are asked to enter. These are stored in str and str1 respectively, where str is a char array and str1 is a string object.

Then, we have two functions display() that outputs the string onto the string.

The only difference between the two functions is the parameter. The first display() function takes char array as a parameter, while the second takes string as a parameter. **This process is known as function overloading.**

Pointers and Strings

A String is a sequence of characters. A string type variable is declared in the same manner as an array type variable is declared. This is because a string is an array of characters type variables.

Since a string is like an array, pointer variables can also be used to access it. For example:

char st1[] = “Pakistan”;

char *st2 = “Pakistan”

In the above statements, two string variables “**st1**” and “**st2**” are declared. The variable “**st1**” is an array of character type. The variable **“st2**” is a pointer also of character type. These two variables are equivalent. The difference between string variables “**st1**” and “**st2**” is that:

- string variable “**st1**” represents a pointer constant. Since a string is an array of character type, the “**st1**” is the name of the array. Also, the name of the array

represents its address its which is a constant. Therefore, **st1** represents a pointer constant.

- string variable "**st2**" represents a pointer variable.

In the following program example, a string is printed by printing its character on by one.

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

void ppp(char *);  // Function Prototype

int main () {
char st[] = "Pakistan";

ppp(st);   // calling function
cout<<"OK";
}
// function definition
void ppp(char *sss)
{
   //loop iterating string using pointer
   while (*sss != '\0')
   {
      cout<<*sss<<endl;
      *sss++;
   }
}
```

Output:

```
P
a
k
```

i

s

t

a

n

OK