Lab 2:

Pointers in C++

In this lab we will be discussing pointers in detail. This is one of the most important concepts in C++ language. Pointers are used everywhere in C++, so if you want to use the C++ language fully you have to have a very good understanding of pointers. They have to become *comfortable* for you.

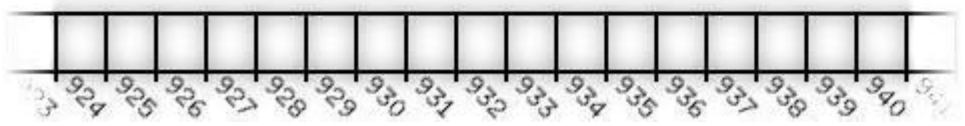
C++ uses pointers in three different ways:

- C++ uses pointers to create dynamic data structures -- data structures built up from blocks of memory allocated from the heap at run-time.
- C++ uses pointers to handle variable parameters passed to functions.
- Pointers in C++ provide an alternative way to access information stored in arrays. Pointer techniques are especially valuable when you work with strings. There is an intimate link between arrays and pointers in C++.

To fully grasp the concept of pointers all you need is the concept and practice of pointers. Before talking about pointers let’s talk a bit about computer memory.

Computer Memory

Essentially, the computer's memory is made up of bytes. Each byte has a number, an address, associated with it. The picture below represents several bytes of a computer's memory. In the picture, addresses 924 thru 940 are shown.



Variable and Computer Memory

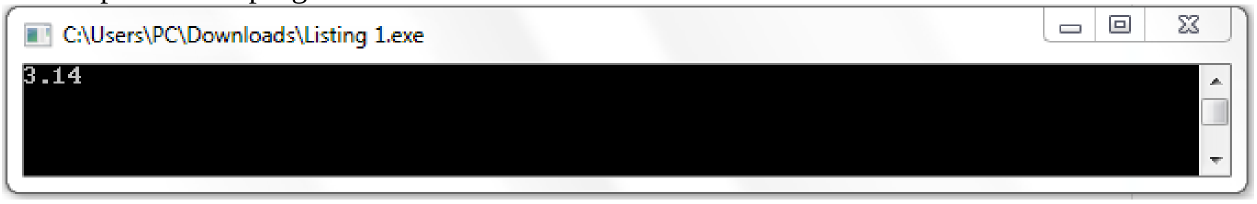
A variable in a program is something with a name, the value of which can vary. The way the compiler handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes.

Example 2.1:

```
#include<iostream>
using namespace std;
int main()
```

```
        {
        float fl=3.14;
        cout<<fl;
        cin>>fl;
        return 0;
        }
```

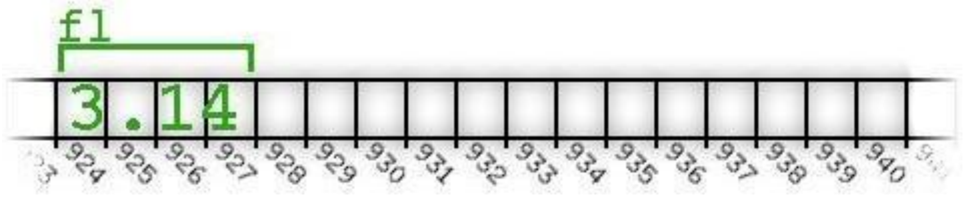
The output for this program will be



```
3.14
```

How It Works: (Explanation)

At line (4) in the program above, the computer reserves memory for fl. Depending on the computer's architecture, a float may require 2, 4, 8 or some other number of bytes. In our example, we'll assume that a float requires 4 bytes.



When fl is used in line (5), two distinct steps occur:

1. The program finds and grabs the address reserved for fl--in this example 924.
2. The contents stored at that address are retrieved

The illustration that shows 3.14 in the computer's memory can be misleading. Looking at the diagram, it appears that "3" is stored in memory location 924, "." is stored in memory location 925, "1" in 926, and "4" in 927. Keep in mind that the computer actually converts the floating point number 3.14 into a set of ones and zeros. Each byte holds 8 ones or zeros. So, our 4 byte float is stored as 32 ones and zeros (8 per byte times 4 bytes). Regardless of whether the number is 3.14, or -273.15, the number is always stored in 4 bytes as a series of 32 ones and zeros.

Pointer:

In C++ a pointer is a variable that points to or references a memory location in which data is stored. A pointer is a variable that **points** to another variable. This means that a pointer holds the memory address of another variable. Put another way, the pointer does not hold a value in the traditional sense; instead, it holds the address of another variable. A pointer "points to" that other variable by holding a copy of its address. Because a pointer holds an address rather than a value, it has two parts. The pointer itself holds the address and that address points to a value.

Pointer declaration:

A pointer is a variable that contains the memory location of another variable. The syntax is as shown below. You start by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Finally you give the name of the variable.

```
Data_type *variable_name
```

Such a variable is called a ***pointer variable*** (for reasons which hopefully will become clearer a little later). In C++ when we define a pointer variable we do so by preceding its name with an asterisk. In C++ we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
   int *ptr;
   int k;
```

ptr is the name of our variable (just as **k** is the name of our integer variable). The '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The **int** says that we intend to use our pointer variable to store the address of an integer.

Referencing Operator

Suppose now that we want to store in **ptr** the address of our integer variable **k**. To do this we use the unary **&** operator and write:

```
ptr = &k;
```

What the **&** operator does is retrieve the address of **k**, and copies that to the contents of our pointer ptr. Now, ptr is said to "point to" **k**.

Dereferencing operator

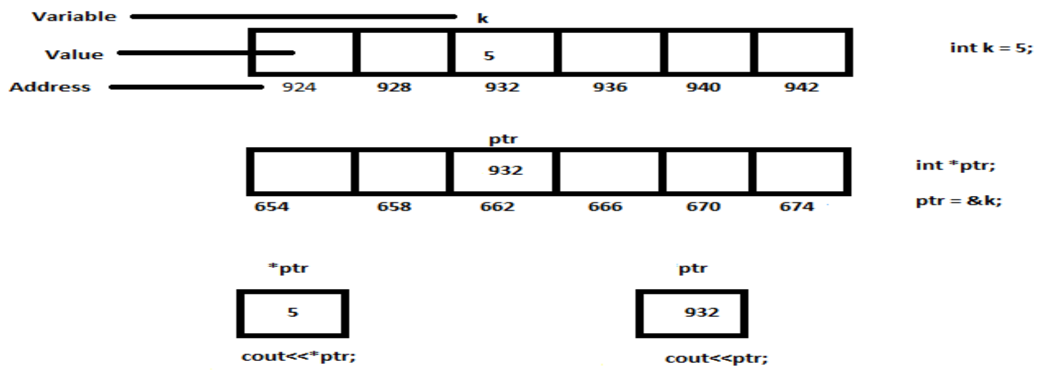
The "dereferencing operator" is the asterisk and it is used as follows:

```
*ptr = 7;
```

will copy 7 to the address pointed to by **ptr**. Thus if **ptr** "points to" (contains the address of) **k**, the above statement will set the value of **k** to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.
Similarly, we could write:

```
cout<<*ptr<<endl;
```

to print to the screen the integer value stored at the address pointed to by **ptr**;.

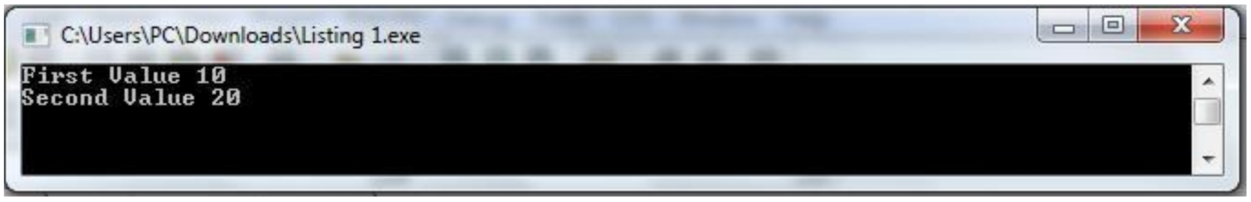


This Example will be very helpful in understanding the pointers. Understand it thoroughly how it works and then proceed.

Example 2.2:

```
#include<iostream>
using namespace std;
int main ()
{
int firstvalue = 5, secondvalue = 15;
int * p1, * p2;

 p1 = &firstvalue;                    // p1 = address of firstvalue
p2 = &secondvalue;                   // p2 = address of secondvalue
*p1 = 10;                            // value pointed by p1 = 10
*p2 = *p1;                          // value pointed by p2 = value pointed by p1
p1 = p2;                            // p1 = p2 (value of pointer is copied)
*p1 = 20;                           // value pointed by p1 = 20
cout<<"First Value is " << firstvalue<<endl;
cout<<"Second Value is " <<secondvalue<<endl;
return 0;
}
```

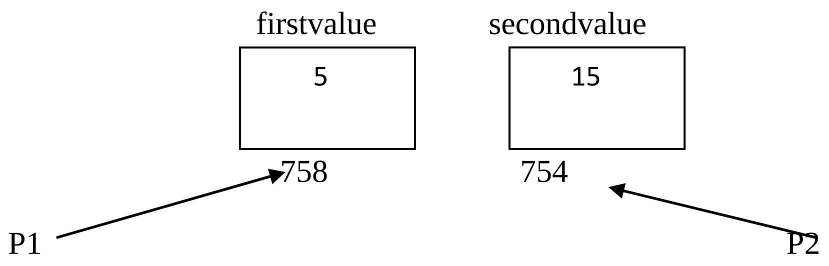
Output of this listing is as follows:



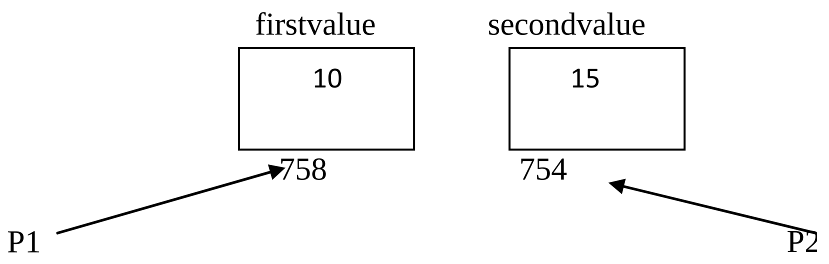
```
First Value 10
Second Value 20
```

How it Works:

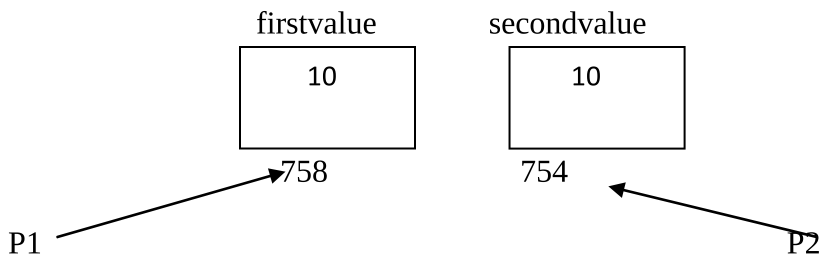
Here in this code we are trying to play with memory and address of our variables for the better understanding of Pointers. On line number 5 we have two integer variables (i.e firstvalue and secondvalue). Both are assigned values of 5 and 15 respectively. On line number 6 we have two integer pointer variables (i.e p1 and p2). Both are assigned addresses of variables in line 5 firstvalue and secondvalue respectively in line 7 and 8.



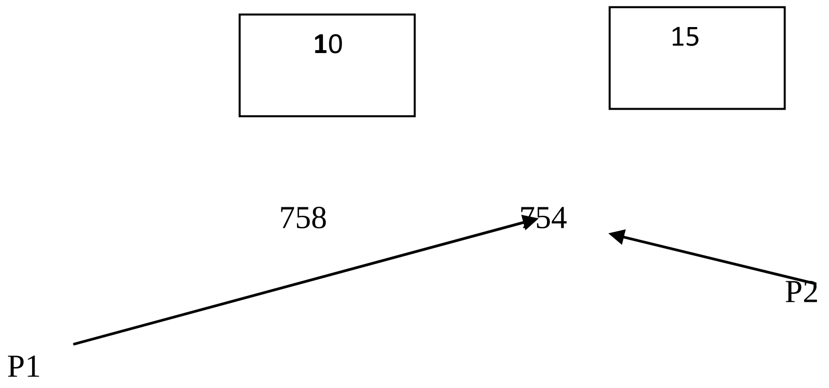
In line 9 we see that *p1 is assigned value 10. This means that 10 should be copied in the variable, which is lying on an address to which p1 is pointing. We know that p1 is pointing to address of firstvalue. So line 9 results in assigning firstvalue the value of 10.



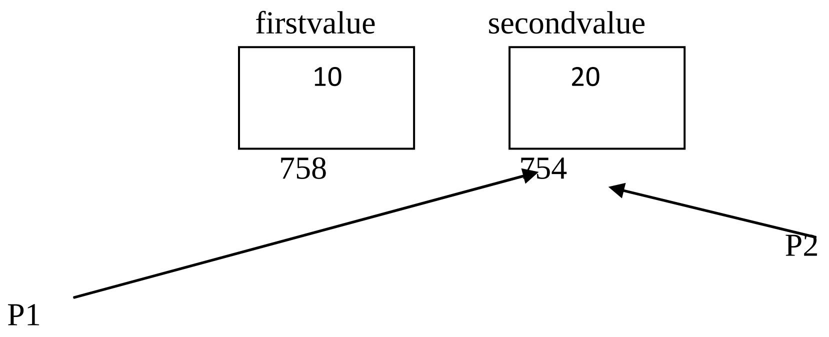
In line 10 we encounter another assignment which says that value of variable pointed by p2 should be replaced with the value of variable pointed by p1. So now secondvalue is assigned with value 10 as well.



Well the assignment in line 11 is a bit confusing but very simple, all this assignment is doing is that now p1 is pointing to the same address as p2. So now we can say p1 and p2 are pointing at same address.



In line 12 we see that *p1 is assigned value 20. This means that 10 should be copied in the variable, which is lying on an address to which p1 is pointing. We know that p1 is now pointing to address of secondvalue because in last line we pointed p1 to the address being pointed by p2. So line 12 results in assigning secondvalue the value of 20.



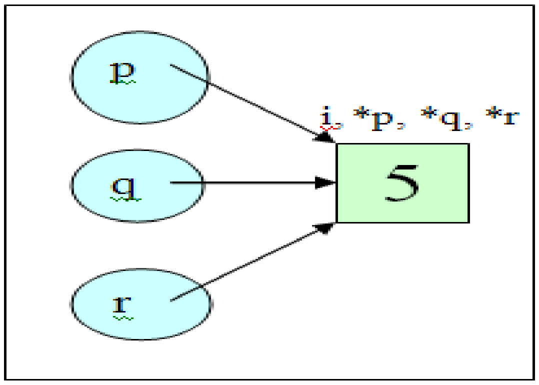
Now when we print the value of first value and second value it prints 10 for firstvalue and 20 for secondvalue; which is right due to the reasons explained above.

Pointers: Pointing to the Same Address

Here is a cool aspect of C++: **Any number of pointers can point to the same address**. For example, you could declare p, q, and r as integer pointers and set all of them to point to i, as shown here:

```
int i;

int *p, *q, *r;



p = &i;

q = &i;

r = p;
```

Note that in this code, **r** points to the same thing that **p** points to, which is **i**. You can assign pointers to one another, and the address is copied from the right-hand side to the left-hand side during the assignment. After executing the above code, this is how things would look:



The variable i now has four names: i, *p, *q and *r. There is no limit on the number of pointers that can hold (and therefore point to) the same address.

Pointer Arithmetic's

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

```
y=*p1 * *p2;
sum=sum+*p1;
z= 5 - *p2/*p1;
*p2= *p2 + 10;
```

C++ allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc.,
we can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

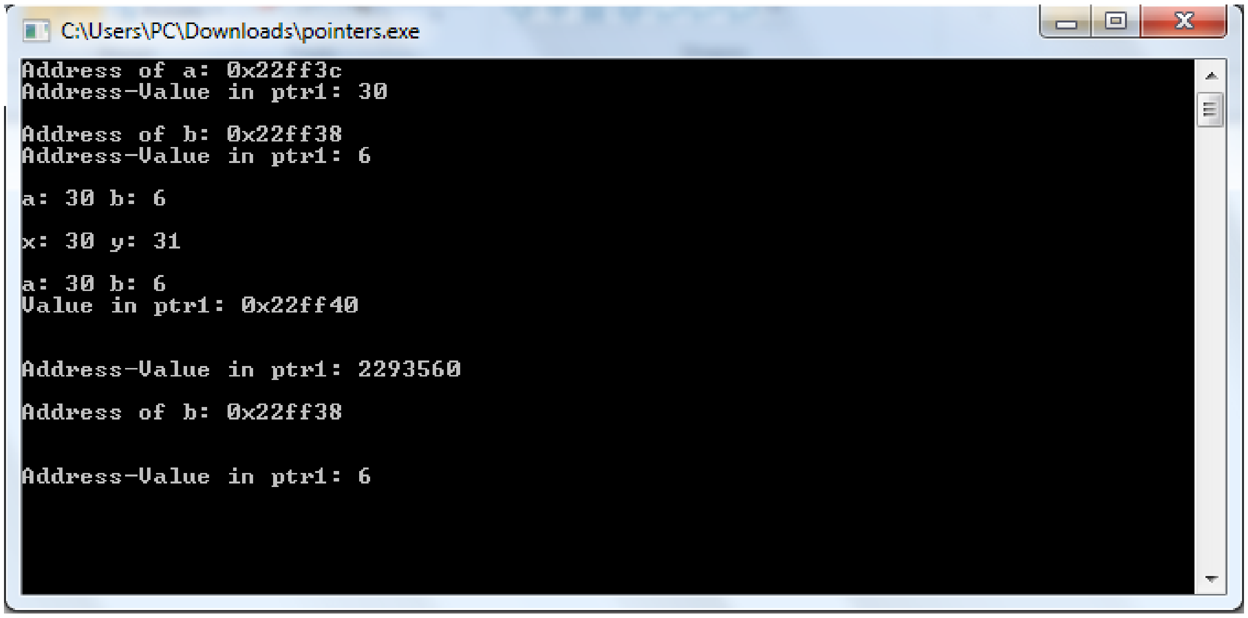
When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers. The number of bytes depends on the object's data type.

/*Program to illustrate the pointer expression and pointer arithmetic*/

Example 2.3:

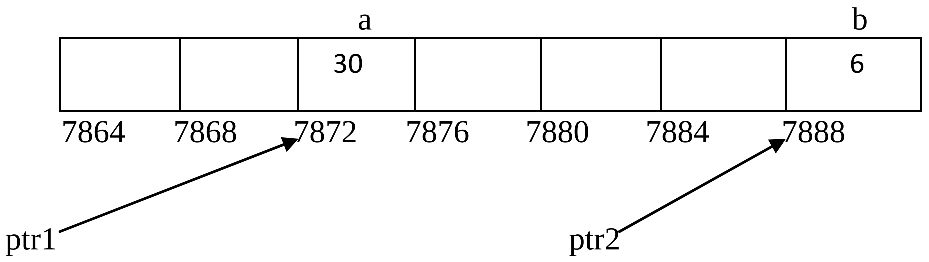
```
#include<iostream>
using namespace std;
int main()
{
int *ptr1,*ptr2;
int a,b,x,y,z;

a=30;b=6;
ptr1=&a;
ptr2=&b;

x=*ptr1 + *ptr2 - 6;
y=6 - *ptr1 / *ptr2 +30;

cout<<"Address of a: "<<ptr1<<endl;
cout<<"Address-Value in ptr1: "<<*ptr1<<endl<<endl;
   //The comment value is the value of ptr1 (address of the a)

cout<<"Address of b: "<<ptr2<<endl;
cout<<"Address-Value in ptr1: "<<*ptr2<<endl<<endl;
   //The comment value is the value of ptr2 (address of the b)

cout<<"a: " <<a<<" b: "<<b<<endl<<endl;
   //Simply prints the value of a and b

cout<<"x: " <<x<<" y: "<<y<<endl<<endl;
   //Simply prints the value of x and y.

ptr1=ptr1 + 1; // adds 4 in address of ptr1. (1*4 = 4)
ptr2= ptr2;
cout<<"a: " <<a<<" b: "<<b<<endl;
   //Simply prints the value of a and b

cout<<"Value in ptr1: "<<ptr1<<endl<<endl; // 2293564
   //The comment value is the new memory location value of ptr1

cout<<"\nAddress-Value in ptr1: "<<*ptr1<<endl<<endl; // garbage value
   //The comment value is the new value of ptr1 (garbage value)
cout<<"Address of b: "<<ptr2<<endl<<endl;
cout<<"\nAddress-Value in ptr1: "<<*ptr2<<endl<<endl;
   cin>>a;
}
```

Here note that adding some thing in *ptr1 changes the value of the address stored in **ptr**. However adding some thing in **ptr** will change the address it is pointing to. Printing **ptr1** after adding 1 in it gives different address as it has changed by 4 Bytes.

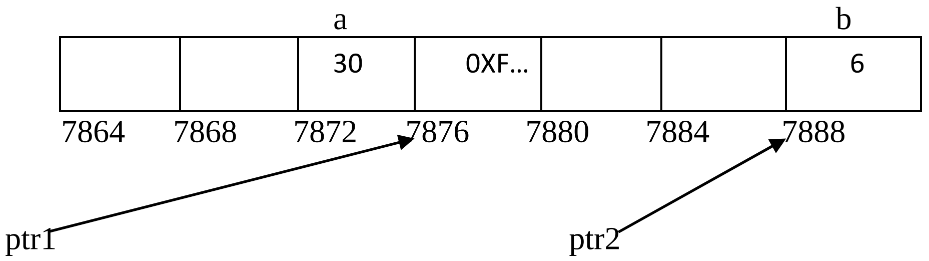


```
Address of a: 0x22ff3c
Address-Value in ptr1: 30

Address of b: 0x22ff38
Address-Value in ptr1: 6

a: 30 b: 6

x: 30 y: 31

a: 30 b: 6
Value in ptr1: 0x22ff40


Address-Value in ptr1: 2293560

Address of b: 0x22ff38


Address-Value in ptr1: 6
```

How it Works:
This code explains all the rules related to arithematic of pointers. From line 1 to line 11, it is simply adding, subtracting and like manipulating with the pointers and variables. After all the manipulations and arithmetics it started printing values of pointers and other simple variables till line 12.



At line 18 it adds 1 to ptr1. Mostly people think that this will change the address of the pointer, but they are totally wrong. Remember pointer is pointing to an address. This addition does not change the address of the pointer, infact it changes the value of the pointer (not the value of the address pointer is pointing at.). **ptr1** has the address of variable of variable **a .** So it adds 1 *4 Btytes = 4bytes in the address of a which is stored in ptr1. Where as ptr2 points at the same value as before due to the assignment of line 19.



Line 20 prints the same value as was printed by the Line 16, because values of the variable was never changed, in fact ptr1's value which was address of a was changed. Now Line 21 will print the value stored in ptr1; which is address of memory 4 bytes ahead of variable **a.** Line 22 is trying to print the value at address, ptr1 is now pointing to, which was never assigned any value.

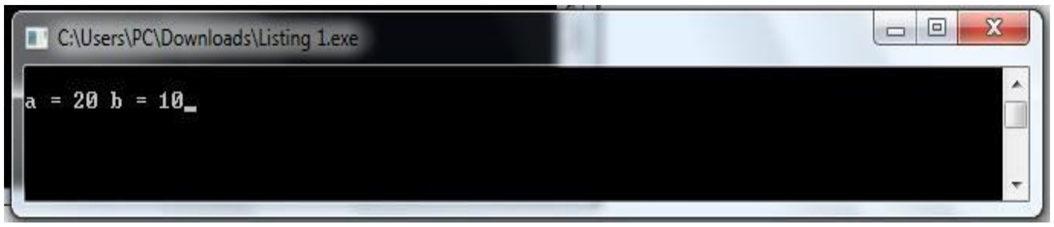
Sending Pointers as Arguments to Functions

When we pass pointers to some function, the addresses of actual arguments in the calling function are copied into the arguments of the called function. This means that using these addresses we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact. **Try this out:**

Example 2.4:

```
        #include<iostream>
        void swap(int *,int *);
        using namespace std;

        int main( )
        {
int a = 10, b = 20 ;
int *p, *q;
p = &a;
q = &b;
swap( p, q) ;

cout<<"a: "<<a<<"b: "<<b<<endl;
cin>>a;
        }
        void swap( int *x, int *y )
        {
int t = *x ;
*x = *y;
*y = t;
        }
```

The output of the above program would be:



```
a = 20 b = 10
```

a = 20 b = 10

Note that this program manages to exchange the values of **a** and **b** using their addresses stored in **x** and **y**.

Function Pointers

A function pointer is a variable that stores the address of a function that can later be called through that function pointer. A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

And simply put brackets around the name and a * in front of it: that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */
int *func(int a, float b); //Wrong

/* pointer to function returning int */
int (*func)(int a, float b);
```

Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. You can call the function as:

```
(*func)(1,2);
```

Example 2.5

```
#include<iostream>
using namespace std;
void func(int);
int main(){
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    cout<<endl;
    fp(2);
  system("PAUSE");
return 0;
}
Void func(int arg)
{
    cout<<arg<<endl;
}
```

Pointers and arrays

The concept of arrays is related to that of pointers. In fact, arrays work very much like pointers to their first elements, and, actually, an array can always be implicitly converted to the pointer of the proper type. For example, consider these two declarations:

```
int myarray [20];
int * mypointer;
```

The following assignment operation would be valid:

```
mypointer = myarray;
```

After that, `mypointer` and `myarray` would be equivalent and would have very similar properties. The main difference being that `mypointer` can be assigned a different address, whereas `myarray` can never be assigned anything, and will always represent the same block of 20 elements of type `int`. Therefore, the following assignment would not be valid:

```
myarray = mypointer;
```

Let's see an example that mixes arrays and pointers:

Example 2.6:

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

int main ()
{
  int numbers[5];
  int * p;
  p = numbers; *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers; *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;}
```

Output would be

```
10, 20, 30, 40, 50,
```

Pointers and arrays support the same set of operations, with the same meaning for both. The main difference being that pointers can be assigned new addresses, while arrays cannot.

In the chapter about arrays, brackets (`[]`) were explained as specifying the index of an element of the array. Well, in fact these brackets are a dereferencing operator known as *offset operator*. They dereference the variable they follow just as `*` does, but they also add the number between brackets to the address being dereferenced. For example:

```
a[5] = 0;        // a [offset of 5] = 0
*(a+5) = 0;      // pointed by (a+5) = 0
```

These two expressions are equivalent and valid, not only if `a` is a pointer, but also if `a` is an array. Remember that if an array, its name can be used just like a pointer to its first element.

Note:
Before doing your lab exercises run the examples.
Exercises will be provided by the instructors in the lab.