Programming Fundamentals CT-175

Pointers

Objectives

The objective of this lab is to familiarize students with the concept of pointers. By the end of this lab students will be able declare, initialize and assign values to pointers, use void and constant pointers, explore relationship between arrays and pointers and pass arguments to functions using pointers

Tools Required

DevC++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Department of Computer Science and Information Technology
NED University of Engineering and Technology

Pointers in C

A pointer is a variable that stores the memory address of another variable as its value. Pointers enable programs to simulate pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.

Declaring a Pointer

Pointers, like all variables, must be defined before they can be used. The definition:
`int *countPtr;`
specifies that variable countPtr is of type int * (i.e., a pointer to an integer) and is read (right to left), "countPtr is a pointer to int" or "countPtr points to an object of type int."

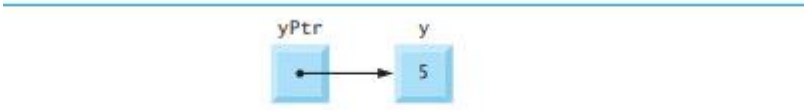
Initializing and Assigning Values to a Pointer

Pointers should be initialized when they're defined, or they can be assigned a value,
The **&**, or **address operator**, is a unary operator that returns the address of its operand.

Example 01: Declaring and assigning value to pointer.

```
#include <stdio.h>
int main( ){
 int y = 5;
 int *yPtr;
 yPtr = &y;
}
```

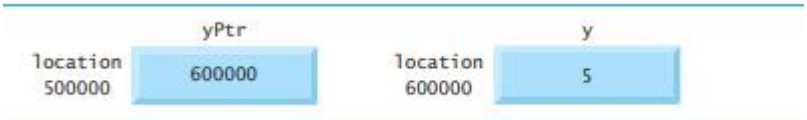
For example, assuming the definitions, the above example assigns the address of the variable y to **pointer variable yPtr. Variable yPtr is then said to "point to" y.**



Graphical representation of a pointer pointing to an integer variable in memory.

Pointer Representation in Memory

The below figure shows the representation of the pointer in memory, assuming that integer variable y is stored at location 600000, and pointer variable yPtr is stored at location 500000. The operand of the address operator must be a variable; the address operator cannot be applied to constants or expressions.



Representation of y and yPtr in memory.

The Indirection (*) Operator

The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand (i.e., a pointer) points. For example, the statement

```
printf( "%d", *yPtr );
```

prints the value of variable y, namely 5. Using * in this manner is called **dereferencing a pointer**.

Void Pointer

The void pointer is a pointer which is not associated with any data types. As the name suggests, the void pointer indicates that the pointer is basically empty- and thus, it is capable of holding any data type address in the program. It is also called general purpose pointer. The syntax:

```
void *ptr;
```

Constant Pointer

A ***constant pointer*** is a pointer that cannot change the address it is containing. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable. These pointers can change the value of the variable they point to but cannot change the address they are holding. Compile and run the below mentioned example. What do you think will happen?

Example 02: Using constant pointer.

```
#include <stdio.h>
int main( ){
 int y = 5;
 int z = 6;
 int const *yPtr = &y;
 printf("%p\n", yPtr);
 *yPtr = &z;
 printf("%p\n", yPtr);
}
```

Pointers and Arrays

Arrays and pointers are intimately related in C and often may be used interchangeably. An array name can be thought of as a constant pointer. Pointers can be used to do any operation involving array subscripting. Assume that integer array b[5] and integer pointer variable bPtr have been defined. Because the array name (without a subscript) is a pointer to the first element of the array, we can set bPtr equal to the address of the first element in array b with the statement

```
bPtr = b;
```

This statement is equivalent to taking the address of the array's first element as follows:

```
bPtr = &b[ 0 ];
```

Array element b[3] can alternatively be referenced with the pointer expression

```
*( bPtr + 3 )
```

The 3 in the expression is the **offset** to the pointer. When the pointer points to the array's first element, the offset indicates which array element should be referenced, and the offset value is identical to the array subscript. This notation is referred to as **pointer/offset notation**. The parentheses are necessary because the precedence of * is higher than the precedence of +. Without the parentheses, the above expression would add 3 to the value of the expression *bPtr (i.e., 3 would be added to b[0], assuming bPtr points to the beginning of the array).

Example 03: Using pointers to access & manipulate array elements.

```
#include<stdio.h>
int main()
{
   int arr[3];

   // declare pointer variable
   int *ptr;
   // declare loop iterator variable
   int i;

   // ptr = &arr[0]
   ptr = arr;

   // use for loop to put values in all array elements using pointer notation
    for (i = 0; i < 3; ++i)
   {
      *(ptr+i) = i+1;
   }

   // use for loop to print values of all array elements using pointer notation
   printf("\nDisplaying value using pointers: ");
   for (i = 0; i < 3; i++)
   {
      printf("%d\n", *(ptr+i));
   }
   // use for loop to print addresses of all array elements using pointer notation
   printf("\nDisplaying address using pointers: ");
    for (i = 0; i < 3; i++)
   {
      printf("%p\n", ptr+i);
   }
   return 0;
}
```

Passing Arguments to Functions by Reference

There are two ways to pass arguments to a function—**pass-by-value** and **pass-by-reference**. All arguments in C are passed by value. As we know, **return** may be used to return one value from a called function to a caller (or to return control from a called function without passing back a value). Many functions require the capability to modify variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).

In C, you use **pointers** and the **indirection operator** to simulate **pass-by-reference**. When calling a function with arguments that should be modified, the addresses of the arguments are passed. This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.

Example 04: Find out cube of a number using passing by reference.

```
#include <stdio.h>

void cubeByReference(int *nPtr) {
        *nPtr = *nPtr * *nPtr * *nPtr;
}
int main( void ) {
        int number = 5; // initialize number
        printf( "The original value of number is %d \n", number );
        //pass address of the number to cubeByReference
        cubeByReference( &number);
        printf( "The new value of number is %d", number );
}
```

Exercise

1. For each of the following, write a single statement that performs the indicated task. Assume that long integer variables value1 and value2 have been defined and that value1 has been initialized to 200000.
   a) Define the variable lPtr to be a pointer to an object of type long.
   b) Assign the address of variable value1 to pointer variable lPtr.
   c) Print the value of the object pointed to by lPtr.
   d) Assign the value of the object pointed to by lPtr to variable value2.
   e) Print the value of value2.
   f) Print the address of value1.
   g) Print the address stored in lPtr. Is the value printed the same as the address of value1?
2. Write a program to implement the exchange or swap the values of 3 variables{a,b,c}. To implement this, you need to write a function called swaped().

   ```
   void swaped(int *aPtr, int *bPtr, int *cPtr);
   ```

   such that; b ----> temp

   a ----> b

   c ----> a

   temp -> a
3. Write a program that calculates the sum of all the elements in array using pointers.
4. Write a program that finds the second highest number in a float type array of 20 elements using pointer.
5. Write a program that implements the SortFunction that takes argument pointer to an array, size of the array and the order in which it is going to be sort. Such as, 1 for Asscending order and 2 for Descending orde. Finally, print this array in Main() to check.

   ```
   void SortFunction(int *arr, int *size, int order);
   ```
6. Declare an array of int at least 10 elements long. Fill in the array with the square of its index using array syntax, a[i] = i * i ;. Print out the array using pointer syntax *(a + i).
7. Consider the following program:

   ```
   #include <stdio.h>
   int main( void ) {
       int *p1;
       char *p2;
       p2=p1;
   }
   ```

   Will the program compile successfully without warnings? Why and why not?