Lab 3: Fun with pointers!

This aim of this lab assignment is to understand pass-by-value vs. pass-by-reference, single/double pointers, dereferencing using pointer variables and understanding an array as a pointer variable.

You are given two .cxx files:

MyClass.cxx:
A toy class consisting of an int variable and basic constructors and getter, setter and print functions.

TestPointers.cxx:
Fix compile time errors in this class (concerning pointer variables) and run this file to see output.

You have in all 4-compile errors to fix (4 points) and few lines of code to explain to your TA (6 points) for a total of 10 points. See comments in TestPointers.cxx for details.

Part 1: Pass-by-value vs. pass-by-reference

```
// Pass-by-value
int incrementValueOf(int x)
{
        return (x+1);
}

// Pass-by-reference
void incrementValueOf_2(int &x)
{
        x ++;
}
```

The function incrementValueOf_2() is an example of passing an integer variable by reference. Passing by reference is also called aliasing. This means that if we call this function from the main function like so:

```
int main ()
{
	int a = 0;
	incrementValueOf_2(a);
	return 0;
}
```

then the variable called as 'a' in main() will be called as 'x' in incrementValueOf_2().

Thus x is an alias (different name for exactly the same thing!) of a.

If we replace the function call of incrementValueOf_2() with incrementValueOf() in the main function such that we have a pass-by-value:

```
int main ()
{
	int a = 0;
	incrementValueOf(a);
	return 0;
}
```

then the variable 'x' in incrementValueOf() function only gets the value 0.

Thus in the pass-by-value case we have two different variables 'x' and 'a' (two different slots in memory), while in the pass-by-reference case both 'x' and 'a' refer to the same slot in memory.

Part 2: Single/Double pointers

Any pointer variable stores some address in the memory. Depending on the type of the pointer we know what 'variable type' resides at that address.

int* x: a pointer variable such that the address value stored in x gives location of an integer variable.

double* y: a pointer variable such that the address value stored in y gives location of an double variable.

int** z: a pointer variable such that the address value stored in z gives location of a single pointer to integer, i.e., 'int*' variable.

Double pointers dereferencing example:

```
int main()
{
	int a = 10;
	int* b  = &a;
	int **c  = &b;

	cout << a <<endl; //Prints 10
	cout << *b <<endl; // Prints 10
	cout << *(*c)<<endl; //Prints 10

	//You can keep playing this game for triple pointers.....n-pointer
	//etc.
}
```

Part 3: Dereferencing components (variables or functions) for a class instance variable.

Suppose we have a some class,

```
class MyClass
{
	public:
		void printHello()
		{
			cout <<"Hello" <<endl;
		}
}
```

Then calling the function printHello() can be done using a pointer to a class instance like so:

```
int main()
{
	MyClass a;
	MyClass* b = &a;

	a.printHello(); // Prints the string "Hello"

	b->printHello(); //Prints the string "Hello"

}
```

Thus we have two operators to access components (variables or functions) within the class instance 'a'.

1. Using the dot , i.e., '.' operator:

This operator expects the class instance on its LHS and hence we do: a.printHello()

2. Using the arrow, i.e., '->' operator:

This operator expects a single pointer to the class instance on its LHS (Left Hand Side) and hence we do:

b->printHello()

The RHS(Right Hand Side) of both operators is the actual component we want to access (in this case the printHello() function).

Note: There is a way we can use the '.' operator and yet use the single pointer 'b' to call the printHello() function.

From **Part 2** we know that '*b' gives essentially the variable 'a' and hence the following works:

(*b).printHello()

Part 4: Understanding an array as a pointer

An array is a set of contiguous locations in memory.

int a[5]: is an array of ints. (Thus we have 5 contiguous slots in memory each having the size of an int)

In this case the variable 'a' refers to the first memory location or the location of the first slot.

Hence 'a' is also a pointer.

The difference between 'a' and say 'int *b':
'a' is a constant pointer and 'b' is non-constant.

(**Note**: Arrays allocated at runtime using the new statement:
int* a = new int[5];
In this case 'a' is also a non-constant pointer.
)

Pointer arithmetic:

1) 'a' or 'a+0' gives address of the first slot

2) 'a+1' gives address of the second slot and so on.

'a+1' gives the second slot because the ' + 1' (plus one) in this case is interpreted as an offset of '1 int' as 'a' is an integer array.

Thus if 'a' was an array of doubles, 'a + 2' is an offset of '2 doubles'.

This is called pointer arithmetic.

Note that from **Part 2,** *(a+1) gives the content inside the second slot. Thus *(a+1) is exactly the same as a[1], which we generally use.