CMSC 202 Spring 2021

Lab 04 – Variables, Passing Variables, and Pointers

Assignment: Lab 04 – Variables, Passing Variables, and Pointers

Due Date: On Discord from February 22nd until February 25th

Value: 6 points

1. Overview

In this lab you will:

- Discuss variables in terms of memory
- Practice passing variables by value and by reference
- Create pointers to practice
- Program both an example of pass-by-reference and simple pointers

2. Introduction

As we have discussed, when you declare a variable you are telling the compiler—and, ultimately, the computer—what kind of data you will be storing in the variable. Each variable requires a specific amount of space to store the related information depending on the data type.

For example, the following are two variable declarations that might occur in a C++ program:

```
int num;
double gradeOne, gradeTwo;
```

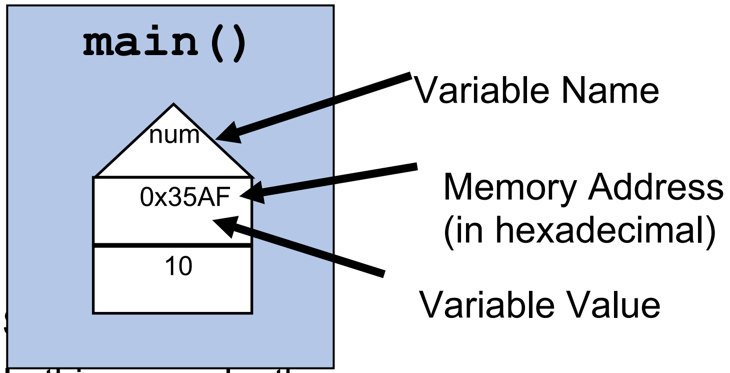
The first line declares a new variable called **`num`** so that it can hold a value of type int. The second line declares **`gradeOne`** and **`gradeTwo`** to be variables of type double, which is one of the types for numbers with a decimal point (known as floating-point numbers).

Behind the scenes, these variables are being stored in memory (RAM) for access later. In modern PCs, the smallest addressable unit is a byte, therefore, when we think of a variable being in a memory space, they usually are in several contiguous bytes of memory. As these program variables are implemented as memory locations, each memory location has a unique address that is a hexadecimal number. The compiler assigns one memory location to each variable.

Now let's look at num again, this time we will declare and initialize it to 10.

```
int num = 10;
```

This variable would look like this:



In this example, the variable **num** has not only a variable name and a variable value but it also has a memory address. The other thing that is important to recognize from the diagram above is that this particular variable exists inside the main function. It is always important to remember that variables that are declared within the body of a function definition are said to be local to that function.

3. Pass-by-Value / Call-by-Value

Normally when we use a function we just pass arguments to the function. These arguments are usually either literals, constants, or variables. Let's look at an example:

```
//Header Information
int main() {
const int NUM_STUDENTS = 100;
float pricePerStudent = 5.50;
float totalPrice = 0.0;
totalPrice = calculateTotal(NUM_STUDENTS,
pricePerStudent, 10);
cout << "The total price is " << totalPrice << endl;
return 0;
}

float calculateTotal(int students,
     float pricePerStudent, int extraStudents) {
return (students + extraStudents) * pricePerStudent;
}
```

In the example above, we have the main function calling the **`calculateTotal`** function. This is the most common way that we call our functions. When control is passed to **`calculateTotal`**, the variables in **`calculateTotal`** are local to **`calculateTotal`**. That is, they are not available in **`main`**. We call these pass-by-value or call-by-value parameters.

In pass-by-value, when the function is invoked, the value of the parameter is calculated, declared, and initialized. Once the function that is called is done running, those call-by-value parameters are deallocated and are no longer available.

From a memory standpoint, when a function is called using pass-by-value or call-by-value, the variables look just like a normal variable declaration and initialization.

In the code above, when **`calculateTotal`** is invoked, it is the equivalent to running this code:

```
int students = 100;
float pricePerStudent = 5.50;
int extraStudents = 10;
```

They are normal variables local to the **`calculateTotal`** function. The variable is accessible via variable name, variable value, and memory address.

Additional information about this topic can be found in your book in chapter 4.

4. Pass-by-Reference / Call-by-Reference

There is another way that we can pass variables to a function other than pass-by-value. What if, instead of passing the variable value to the function, we passed it the memory address?

Memory addresses are not limited by scope like a variable is. This is because we are accessing a variable by a memory address which is not subject to the same scope limitations that a variable name is. So, if we are in a function that has the memory address of a variable in another function then we can access that variable by accessing the memory address of the variable.

In order to take advantage of this situation, we can pass the memory address to a function by using an ampersand (`&`). We call this pass-by-reference or call-by-reference. For example:

```
//Header Information
int main () {
  int myAge = 0;
  getAge(myAge); //This is a normal void function call
  cout << "Your age is " << myAge << endl;
return 0;
}

void getAge (int &age) { //Notice that we have an &
  cout << "Enter your age: ";
  cin >> age;
}
```

Notice in this example that **`getAge`** is a void function which means that it does not return anything. Additionally, notice how we passed the argument **`myAge`** to the **`getAge`** function using an ampersand. This is an example of pass-by-reference or call-by-reference. In this example, we pass the **`myAge`** variable to the function **`getAge`**. Rather than taking the variable value (which is what pass-by-value does), we take the memory address. Because we have the memory address in the function, any changes to the variable **age** in

getAge will also change the value of **myAge** in main. *Read that last sentence again, it is very important.*

By using pass-by-reference or call-by-reference, it allowed us to change the value of a variable that was declared in main inside of another function. By passing the variables by memory address, we can change the value of many variables without using the return keyword.

Additional information about this topic can be found in your book in chapter 4.

5. Pointers

Now that we have looked at pass-by-value and pass-by-reference, we can take it one step further and we can look at memory addresses in more detail. A pointer is the memory address of a variable. To be honest, when we do pass-by-reference, we are just passing the function a pointer rather than the variable.

A pointer variable can be declared by including the asterisk or star (*) in the variable declaration. When we declare a pointer variable, we need to make sure that the data type of the pointer variable matches the data type of what the pointer is going to point to. For example, if we are going to make an integer variable and a pointer to the integer variable, it might look like this:

```
int students = 100;
int *ptrStudents;
ptrStudents = &students;
```

In the example above, we have declared an integer variable named **students** that is initialized to 100. Then we declare an integer pointer that doesn't point to anything initially. Finally, in the third line, we set the pointer variable to the memory address of students. Notice that the pointer is set to **&students**. This is because if we use an & with a variable, it allows us to access the memory location of the variable.

If we wanted to access the variable value of students using the pointer, we can't just use **ptrStudents** because that variable's value is the memory address for students. If we want to access the variable value, we need to use the asterisk or star (*) again. This is called *dereferencing* the pointer.

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

int main(){
  int students = 100; //Declare and initialize var.
  int *ptrStudents; //Declare pointer
  ptrStudents = &students; //Initialize pointer
  cout << *ptrStudents << endl; //Dereference pointer
}
```

So, we use the ampersand (&) anytime we want to access the memory address of a variable. The ampersand is called the *address of* operator. We use the asterisk (*) to declare a pointer or anytime we want the variable value of what the pointer is pointing at (dereferencing).

The other use for the asterisk is if we want to use the pointer to change the value of what the pointer points to. Here is an example:

```
int main(){
  int students = 100; //Declares + initializes var.
  int *ptrStudents; //Declares pointer
  ptrStudents = &students; //Init. pointer to memory
  cout << "pointer before: " << *ptrStudents
       << endl; //Dereferences pointer
  *ptrStudents = 50; //Assigns dereferenced pointer
  cout << "pointer after: " << *ptrStudents
       << endl; //Dereferences pointer
  cout << "students after: " << students
       << endl; //Students
}
```

At the end of this code running, **`students`** will have been changed to 50 because we used the dereferenced pointer to change the value of the **`students`** variable.

Additional information about this topic can be found in your book in chapter 10.

6. Prelab Quiz

- When you have finished reading over this prelab, you need to complete your prelab quiz on Blackboard under "Assignments" then "Labs" then Prelab. You have until Monday at 10am to complete your prelab quiz.