**Name:**

**Advanced Programming in C++**

**Lab Exercise 3/29/2021**

In this lab, you will:

* See the definition of a pointer.
* Observe the basic pointer operators.
* Master the mechanism for simple pointer manipulation.
* Realize how to use pointers to pass parameters by reference.
* Learn how pointers are used in conjunction with dynamic data.
* Discover what happens when a pointer goes into the unknown!
* Complete programs that work with pointers.

Lab Exercise:

* In the first part of the exercise, you will print out what happens with simple pointer variables.
* In the second part, you will create several functions that utilize pointers and references.
* In the third part, you will create a dynamic array that you will use.

**Definition of a Pointer.**

Pointers are a type of variable that allow you to specify the address of a variable. They provide a convenient means of passing arguments to functions and for referring to more complex data types such as structures. They are also essential if you want to use dynamic data in the free store area. (That was a free look ahead to the dynamic data topic covered later in these notes.)

You won't always know the specific value in a pointer, but you won't care as long as it contains the address of the variable you are after. You need to declare and initialize pointers just as you would other variables, but there are special operators that you need to use.

**Pointer Operators**

Here is a table showing the special characters used in C++ to declare and use pointers.

|  |  |  |
| --- | --- | --- |
| **\*** | dereferencing operator,  indirection operator | This is used ***to declare*** a variable as a pointer.  It is also used when you want ***to access*** the value pointed to by the pointer variable. |
| **&** | reference operator,  address-of operator | Use before a variable to indicate that you mean the ***address of*** that variable. You'll often see this in a function header where the parameter list is given. |
| **->** | member selection operator | This is used to refer to ***members of structures*** |

**Simple Pointer Use**

We'd better look at some examples to make this clear.

First, we'll declare two ordinary integers, and also pointers to those integers.

int alpha = 5;

int beta = 20;

int \*alphaPtr = &alpha;

int \*betaPtr = &beta;

The characters Ptr in the pointer variable name have no special significance. They are simply a memory aid for the programmer. Let's look more closely at one of the pointer declarations.

int \*alphaPtr = &alpha;

The first part int \*, tells the compiler to declare a pointer to integers. alphaPtr will be the name of that pointer. In the last part of that statement, &alpha; specifies that the address of the variable alpha is what should be assigned to the pointer variable.

An aside here: It is also permissible to position the asterisk closer to the pointer variable name, i.e. int \*alphaPtr. However, the convention seems to be moving towards placing the asterisk closer to the datatype.

Try to visualize memory after these declarations, thinking of the pointer variable as not having a particular value, simply links to the variables to which they had been assigned.

Now let's look at a trivial example of how to access this data.

\*alphaPtr += 5;

\*betaPtr += 5;

After these statements, only the contents of the alpha and beta variables are changed.

You might say, "What's the big deal here? I could just as easily have written this:"

alpha += 5;

beta += 5;

True, but typically pointers aren't used in such a simple manner. This just illustrates the proper use of pointer syntax.

A great way to keep track of pointers (and avoid getting lost) is to draw a memory map. E.g., another way to approach alpha and alphaPtr above is to think of how the compiler treats them. When you first declare int alpha = 5;, the first thing the compiler does is setup a memory map as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | alpha | alphaPtr |  |
| **Value** | 5 | 0x03 |  |
| **Address** | 0x03 | 0x09 | ... |

The compiler then:

* Gets some space in memory where it can store an integer
* Binds (or associates) the name, alpha, with that memory space
* Finally, it stores the value we initialized alpha to (i.e., stores the value 5) at that memory address

Next, when we declare and initialize alphaPtr (via int \* alphaPtr = &alpha;), it does the same thing:

* Gets some space in memory where it can store a pointer to an integer
* Binds (or associates) the name, alphaPtr, with that memory space
* Finally, it stores the value we initialized alphaPtr to (i.e., stores the value 0x03) at that memory address

Normally, when you do something like cout << alpha;, the compiler does a couple of things: first it looks up the name alpha, then it finds the address associated with that name, and then it finds the value and prints it out (i.e., it prints out the value 5).

Now, when we dereference a pointer using the indirection operator (as in, cout << \*alphaPtr;), the compiler does the opposite: it first looks up the address stored as the value of the pointer variable (i.e., the value of alphaPtr, which is 0x03); then, it finds the value stored at that address and prints this value out (i.e., it again prints out the value 5).

**Using Pointers and References to Pass Parameters.**

A more realistic example of pointer use is to see how pointers (and references) can be used in passing parameters to a function by address (and by reference). In other words, you want to pass the addresses of the data to a function rather than the values of the data when passing by address (also referred to as pass by pointers).

Let's look at both pass by value and pass by address in addition to pass by reference to make sure we understand the difference between all three. To illustrate how parameters are passed by value, let's take a look at the following:

|  |
| --- |
| #include <iostream>  using namespace std;  void exchange (int x, int y);  int main ( )  {  int a = 5;  int b = 9;  cout << "This program attempts to exchange two values." << endl;  cout << "Values before the exchange:" << endl;  cout << "a= " << a << " b= " << b << endl;  exchange(a, b); // code that calls the function  cout << "Values after the exchange:" << endl;  cout << "a= " << a << " b= " << b << endl;  return 0;  }  // function for passing by value  void exchange (int x, int y)  {  int temp;  temp = x;  x = y;  y = temp;  } // end exchange |

Now let's compile this code and run it... did the values of a and b change (please answer Yes or No)? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The values of a and b in the main program have not been changed. If that is what you really wanted to do, you should have used pointers to pass the parameters by address (pointers) or references to pass the parameters by reference.

To do this via pass by address (or pointers), you need to change the function prototype and header to:

void exchange(int \*, int \*);

And also change how the function is called within main() to pass the addresses of the actual parameters, as in:

exchange(& a, & b);

Note, you also have to de-reference the formal parameters within the function definition since the actual parameters passed in were actually addresses (this is why we used the address-of operator in the function prototype and header). Here is what the program should look like now:

|  |
| --- |
| #include <iostream>  using namespace std;  void exchange (int \* x, int \* y);  int main ( )  {  int a = 5;  int b = 9;  cout << "This program exchanges 2 values." << endl;  cout << "Values before the exchange:" << endl;  cout << "a= " << a << " b= " << b << endl;  exchange(&a, &b); // code that calls the function  cout << "Values after the exchange:" << endl;  cout << "a= " << a << " b= " << b << endl;  return 0;  }  // function for passing by address (or pointers)  void exchange (int \* x, int \* y)  {  int temp;  temp = \*x;  \*x = \*y;  \*y = temp;  } // end exchange |

To do this via pass by reference, you need to change the function prototype and header to:

void exchange(int & x, int & y)

Note how you don't have to de-reference the formal parameters within the function definition and you didn't have to add the address-of operator to the actual parameters when you called the function within main(). All of that is done implicitly and automatically for you in the background when you use references! Here is what the whole program looks like now:

|  |
| --- |
| #include <iostream>  using namespace std;  void exchange (int& x, int& y);  int main ()  {  int a = 5;  int b = 9;  cout << "This program exchanges 2 values." << endl;  cout << "Values before the exchange:" << endl;  cout << "a= " << a << " b= " << b << endl;  exchange(a, b); // code that calls the function  cout << "Values after the exchange:" << endl;  cout << "a= " << a << " b= " << b << endl;  return 0;  }  // function for passing by reference  void exchange (int& x, int& y)  {  int temp;  temp = x;  x = y;  y = temp;  } // end exchange |

Now, when the function is executed, the values of a and b **will** be changed in the main program.

**Dynamic Data and Pointers.**

Dynamic data items are called dynamic because they are created and deleted at run time. There are two operators used to perform these functions.

|  |  |
| --- | --- |
| **new** *datatype* | Used to allocate a dynamic variable.  e.g. int \*TmpPtr = new int; |
| **delete** *pointer* | Used to deallocate a dynamic variable.  e.g. delete TmpPtr; |

The new operator is used to create dynamic data variables in the so-called free store, available in memory for this purpose. Free space is also referred to as the heap. A couple of points to remember when you use the new statement.

1. You can't directly name a dynamic data variable as you can other regular variables (this is why dynamically-allocated variables are also called anonymous variables). If you can't name it, then how do you reference it? The answer is that you do this with a pointer.
2. When the new operation is executed, it returns a pointer to the location of the variable space in the free store.
3. If new was unsuccessful in allocating enough memory for a new variable, it will return the value null zero as the value of the pointer variable.

So let's look again at that example of using new.

int\* TmpPtr = new int;

The first part, int\* TmpPtr, declares an integer pointer named TmpPtr. The second part, new int, creates a space in the free store, and returns a pointer to that space. The returned address is assigned to TmpPtr. This is typical of C++; i.e. you can accomplish a lot in a single line of code.

You can declare arrays in the free store as well as simple variables.

int\* WeightPtr = new int [3];

To use this dynamic array element you could code:

WeightPtr[1] = 17; // Note: the "\*" is not needed in an array reference.

//But since an array name is simply a

//constant pointer to the starting

//address of that array, you can also

//access the 2nd element with:

\*(WeightPtr + 1)

We'll see more of this in the programming exercise for this week's lab exercise. The whole idea of using dynamic data is to economize on memory space. So when you've finished with a piece of dynamic data you should release the space with the delete statement.

delete TmpPtr;

delete [] WeightPtr;

This releases space in the free store, but does not delete the pointer. Be careful here! If you try to use the pointer again, after the delete statement, you don't know what address will be in the pointer. Beware the dreaded segmentation fault, core dump! (See the next section for details.)

To safeguard an inadvertent overwrite of a critical area in memory, it is advisable to set pointers to NULL after you delete the associated dynamic data space.

TmpPtr = NULL;

WeightPtr = NULL;

**Lab Exercise -- C++ Pointers**

The purpose of these exercises is to provide a level of comfort with the following tools:

* Passing parameters to a function by address (pointers) and by reference.
* Creating and using dynamic data arrays.
* Combining pointers, structures, and dynamic data arrays.

**Problem 1**

Examine the program *pointers.cpp*. Deduce the values that will be displayed by the cout statements and enter them in the table below. Enter and test the code to check.

|  |  |  |  |
| --- | --- | --- | --- |
| **Line** | **\*p1** | **\*p2** | **x** |
| 16 |  |  |  |
| 19 |  |  |  |

|  |
| --- |
| /\* pointers.cpp, demonstrate use of pointers \*/  #include <cstdlib>  #include <iostream>  using namespace std;  int main( )  {  int x, \*p1, \*p2;  x = 1;  p1 = new int;  \*p1 = 5;  p2 = new int;  \*p2 = 3;  cout << "p1 is " << \*p1 << "\np2 is " << \*p2 << "\nx is " << x << endl; // Line 16  x = \*p2;  p1 = &x;  cout << "p1 is " << \*p1 << "\np2 is " << \*p2 << "\nx is " << x << endl; // Line 19  return 0;  } |

**Problem 2**

Now let's try to hone our skills at both passing by pointers and passing by reference and, at the same time, compare these two methods to the more basic pass by value. You will do this by modifying the program parameters.cpp. To do so, first create a new file called parameters.cpp in Visual Studio and enter the following code. Then, modify the source to add code to each of the function stubs (function skeletons) so that they perform the specified functionality. Finally, enter cout statements and calls to these functions in main( ) to test drive each of these functions (follow the example given for the add function). Finally, compile the program and ensure the validity of the output.

|  |
| --- |
| /\* parameters.cpp,  demonstrate calling functions by value, by pointers, and by references \*/  #include <iostream>  using namespace std;  //Function Prototypes  int add(int, int) ;  void swap(int \*, int \*);  void dec(int &);  void inc(int &);  int main()  {  int num1, num2;  int result;  cout << "\nEnter two numbers separated by a space: " << endl;  cin >> num1 >> num2;  cout << "\n num1 is " << num1 << ", num2 is " << num2 << endl;  /\* Alter and complete the rest of the code to  test the add, swap, decrement, and increment functions and  also display each result \*/  // result = add(num1, num2);  // cout << "\n result is " << result << endl;    // Add Code Here  return 0;  } //end of main  // add two numbers together and return the result  int add(int x, int y)  {  // Add Code Here  }  //take two pointer references to integers and swap their values  void swap(int \*x, int \*y)  {  int temp;  // Add Code Here  }  //takes one integer by reference and decrements the value pointed to by x  void dec(int &x)  {  // Add Code Here  }  //takes one integer by reference and increments the value pointed to by x  void inc(int &x)  {  // Add Code Here  } |

**Create a dynamic array and use it directly.**

This part of the exercise gave you the opportunity to declare and use a dynamic data array. Now that you've demonstrated your ability to pass pointers and references to functions, let's get into using some dynamic data along with this.

Work in the main( ) function only here — don't worry about new functions. Create a new file in Visual Studio called student\_scores.cpp and do the following:

|  |
| --- |
| **Array of student scores declared as dynamic data:**  Declare a dynamic data array to contain the scores for 10 students. Then, step through the array assigning the counter as the value of the score for that array element. Finally, write another for loop that displays each score. Remember that you have to use the pointer to access the array. You can do this in three steps:   1. Declare a pointer and allocate space for your array:   int \* arr = new int[SIZE];     1. Use a for loop to step through the array (using either *pointer notation* or the *subscript notation*):   for (int i=0; i<SIZE; i++) {  arr[i] = i;  }     1. Finally, use another for to display the score:   for (int i=0; i<SIZE; i++) {  cout << "Score for student " << i + 1 << ": " << \*(arr + i) << endl;  }    Don't forget to use delete to free up the space used by the array! |

Compile and run this C++ program and ensure the validity of the output (See Dynamic Array Demo).

**One Final Challenge**

Write a program to determine the longest non-decreasing sequence in a list of randomly generated integers using an array of 100 randomly generated integers in the range of 1 to 100 to serve as your list. Use pointers to keep track of your current non-decreasing sequence and the current longest non-decreasing sequence.