**Andrew Tran**

**Lab Experience Three**

**Objectives:**

1. To work with arrays
2. Understanding memory management within a programming environment.
3. Understanding what happens when bounds checking is not performed.
4. Introduction to pointers.

**Background**

Given the declaration:

**int arrayName[10];**

this allocates 10 storage locations of type **int** in memory. To access each element of the array a subscript must be used. For example, **arrayName[2]** will access the third element of the array called **arrayName**. Memory is allocated in the computer as indicated below:

0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| arrayName |  |  |  |  |  |  |  |  |  |  |

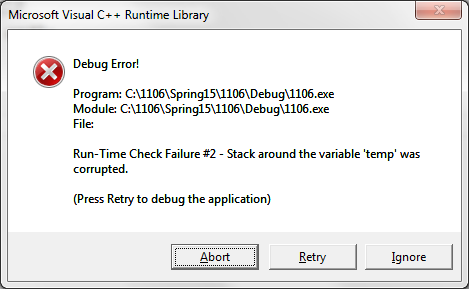
arrayName will contain the starting address of the array and the subscripts will be used as an offset into the actual array elements.

Given **arrayName[2]= 3;**the compiler translates this to: \*(arrayName + 2 \*sizeOf(int))= 3;

Where 2 is the subscript or offset into the array, arrayName contains the starting address, and sizeOf will return the number of bytes the data type is allocated for the current computer system where the program is executing.

**Bounds Checking**

C++ does not perform bounds checking on arrays. If the subscript is not within the boundaries specified other memory locations will be overwritten without the programmers’ knowledge. When using Visual Studio the following debug error message will appear basically notifying you the subscript is out of range.



We will be causing the above window to appear on purpose to examine what happens when subscripts are out of bounds. For the remainder of this lab click on the ignore button to keep processing/running your program.

Given the code below:

#include <iostream>

using namespace std;

const int SIZE = 10;

int main(){

int temp[5] = {0};

int a[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

for(int i = 0; i < 15; i++)

a[i]= i + 1;

for(int i = 0; i < 5; i++)

cout << "The value in temp[" << i << "] is " << temp[i] << endl;

return 0;

}

Notice the first for loop exceeds the bounds of the array **a** by a value of five. When the above program executes the following is output for the contents of the array **temp**:

**The value in temp[0] is 13**

**The value in temp[1] is 14**

**The value in temp[2] is 15**

**The value in temp[3] is 0**

**The value in temp[4] is 0**

Notice: The first three elements are changed because the first for loop exceeded its bounds by a value of five.

Why does this occur? Examining the addresses of all the variables, i.e. a and temp used in the program we can create a memory map to determine what memory locations will be changed.

Below is a memory map for the array a plus 5:



Below is memory map for temp:



Noticed the shaded areas have the same memory addresses.

It is also apparent Visual studio “pads” the area around the arrays by two extra storage locations.

**Background on Pointers**

Variables consist of four components in a computer system and they are:

1. A variable’s name is the way we normally refer to it in a program. Example: **int x;**
2. The variable’s address is the memory location associated with its name.
3. The variable’s type indicates the kind of value to be stored in its memory location, which in turn determines its size, or the number of bytes needed for the variable.
4. The variable’s value is the contents of its memory location.

**What is a pointer?**

A pointer is a variable that can **only** contain an address of a memory location.

**How do you declare pointers?**

A pointer variable is preceded by the dereferencing operator, which is **\***, as follows

**int \*ptr;**

**ptr** is an integer pointer variable. It can only contain addresses of integer data types. I.e. type int. **Note: A common mistake is to assume the variable already contains an address, remember if you don't assign a value to a variable before using it, the contents of the variable is garbage.**

**How do you assign addresses to pointers?**

To assign an address to a pointer you utilize the address of operator, which is **&**, for any variable except arrays and other pointer variables. Why? The array name is a pointer variable and already contains the starting address of the array. Example:

**int \*ptr; // pointer of type integer can only contain addresses**

**// of integers**

**int x = 3; // a regular variable**

**int a[100]; // a integer array**

**double \*dptr; // type double pointer, can only contain addresses of**

**// doubles**

**double y = 2.5; // primitive variable**

**double b[100]; // a double array**

**ptr = &x; // store the address of x into ptr;**

**dptr = &y; // store the address of y into dptr;**

**ptr = a; // store the starting address of the array a into ptr;**

**dptr = b; // store the starting address of the array b into dptr;**

**ptr = dptr; // Error why? Number of bytes assigned to variables of type**

**// double and int are not equal.**

**De-referencing Pointers**

De-referencing pointers means to access the contents of the memory location whose address is contained in the pointer variable. This is accomplished by the dereferencing operator. Example:

**int \*ptr; // pointer of type int can only contain addresses of integers**

**int x = 3; // a regular variable**

**int a[100]; // a integer array**

**double \*dptr; // pointer of type double can only contain addresses of doubles**

**double y = 2.5; // primitive variable**

**double b[100]; // a double array**

**ptr = &x; // store the address of x into ptr;**

**cout << \*ptr**

**<< endl; // will display the contents of the variable x. Why?**

**// Because the address of x was assigned to the variable ptr.**

**\*ptr = 5; // changes the contents of x to the value of 5.**

**int \*dummyptr; // declare another pointer**

**cout << \*dummyptr << endl; // will generate an error. Why? nothing was assigned**

**// to dummyptr**

When the compiler processes the following declarations

**char ch;**

**int intVal;**

it allocates(or sets aside) memory for the variables **ch** and **intVal**. If the memory location set aside for **ch** is **0x08**, and the compiler allocates **ch** and **intVal** in adjacent memory locations, then we might picture memory as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| .... | **0x08** | **0x09** | **0x0A** | **0x0B** | **0x0C** | .... |
|  |  |  |  |  |  |  |
|  | **ch** | **intVal** | | | | |

Such a picture is called a memory map, because it represents a mapping between a program’s variable names and its memory addresses, which typically are represented in hexadecimal (base-16) notation. Note that the memory address associated with **intVal** is **0x09**, even though **intVal** actually consists of locations **0x09** through **0x0C**, as indicated by the shaded part of the picture.

A variable’s name can be thought of as symbolic replacement for its address, because an access to a variable is really an access to its memory location. To illustrate, assigning a value to a variable,

**ch = ‘A’**

simply changes the value of the variable’s memory location. If ASCII code is in use, we can picture the result of such an assignment as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| .... | 0x08 | 0x09 | 0x0A | 0x0B | 0x0C | .... |
|  | **65** |  |  |  |  |  |

**Constant Pointers and Arrays**

Constant pointers are pointers whose contents cannot change. An array name is a constant pointer, which allows greater flexibility in programming.

Given the following declaration:

int xArray[10]; // declares 10 storage locations

Since xArray is a pointer, pointer dereferencing can be used to access each memory location.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **xArray** |  |  |  |  |  |  |  |  |  |  |

What are the contents of **xArray** after execution of the statements below?

**\*xArray = 5;**

**\*(xArray + 4) = 10;**

**Answer**:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **xArray** | 5 |  |  |  | 10 |  |  |  |  |  |

This means \*xArray is the same as xArray[0] and \*(xArray + 4) is the same as xArray[4]. The reason pointers are used instead of subscripted variable is to increase the speed of execution bypassing the translation step.

**Pointer Arithmetic**

Arithmetic operations may be performed on pointers. The operations are:

1. ++ and –
2. A constant value may be added or subtracted from a pointer variable. The previous address stored in the pointer is destroyed in favor of a new address.
3. Pointers can be subtracted from other pointers.

Example:

int xArray[10]; // declares 10 storage locations

int \*xptr = xArray; // store the address of xArray into xptr

// notice the absence of & in front of xArray

for(int i = 0; i < 10; i++){

\*xptr = 5 \* i;

xptr++;

}

Answer:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **xArray** | 0 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |

**Dynamic Memory**

Every array that has been declared is considered a static allocation of memory. This means the compiler places a request to the operating system for memory before execution of the program begins. It is possible to allocate memory during program execution. This process is called dynamic memory allocation. To allocate memory during execution the **new** operator is used and a request is made to the OS for more memory. The OS can either grant the request or deny the request. If the request is denied a NULL reference is stored into the pointer.

An example of dynamic memory allocation is shown below:

**int \*ptr;**

**ptr = new int [10]; // allocates an array of 10 storage locations**

**if(ptr == NULL){**

**cout << “\n\n Error allocating memory --- terminating\n\n”;**

**exit(1106); // error out with code 1106**

**}**

**for(int i = 0; i < 10; i++)**

**ptr[i] = 5 \* i; // can use subscripts to reference each memory location.**

**// or can use pointer offset \*(ptr + i)**

**// But you don’t want to change ptr, otherwise a memory leak will occur.**

**delete [] ptr; // when done, return memory back to the OS**

**Lab Exercises**

**Directions:**

Start Microsoft word and record the questions and answers to all of the exercises in this document.

**Exercise 1**

**Directions:**

1. Download the program pointers.cpp from D2L.
2. Create a new project and a new source file called pointers.cpp.
3. Using the ***address of*** operator add the statements to ***pointers.cpp*** to display the addresses of **int1, int2, int3, dub2, and dub3,** and record their addresses below. i.e. **cout << &int1**, etc. Draw a memory map (similar to the above) of the memory allocated to the six variables.
4. There are situations in which it is useful to define a variable whose purpose is to store an address. Variables that only contain addresses are called pointer variables or simply pointers. Their values lead or point to other addresses.

One of the uses of a pointer variable is to hold the address of another variable. Because variables can be of different types and different types are of different sizes, a pointer variable must be declared as a pointer to a type. The general declaration notation is:

**datatype \*pointerName;**

declares a **pointerName** as a variable capable of holding the address of a variable of **datatype**. For example:

int \*p; // means p can only contain address of integer memory locations

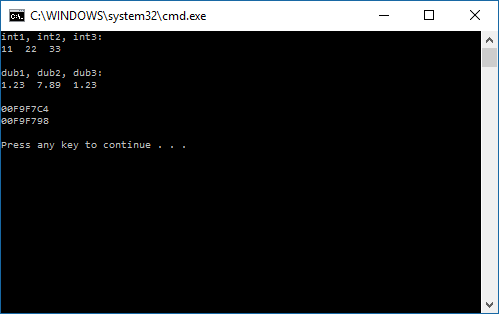
char \*q ; // means q can only contain an address of character memory locations.

**Note: The \* precedes the variable name and is used by C++ to distinguish it as a pointer. A pointer can only contain addresses of other memory locations.**

Add the declarations to declare two **int** pointer variables **intPtr1** and **intPtr2** and two double pointer variables **dubPtr1** and **dubPtr2** to the program **pointers.cpp**.

Note: The memory reserved for pointer variables is 4 bytes for a 32 bit operating system or 8 bytes for a 64 bit operating system.

1. Add statements to your program to assign the address of **int2** to **intPtr2** and address of **dub2** to **dubPtr2**. Add additional statements to your program to display the values of **intPtr2** and **dubPtr2**. Record the output below.



1. Is the value of **intPtr2** the same as the address of **int2** that you recorded part A?

**Yes**

1. Is the value of **dubPtr2** the same as the address of **dub2** that you recorded in part A?

**Yes**

1. Write the statement to assign **intPtr2** to **intPtr1**. Compile and execute and report the output product:

**intPtr1 is uninitialized**

1. Assign **intPtr1** to **dubPtr1** and report what happens. Did it work? Why or why not?

**No, datatypes must match.**

1. Change the statement to: **dubPtr1 = static\_cast<(double \*)>(intPtr1);** and report what happens. Did it work? Why or why not?

**No, you can’t convert from int to double.**

1. What can you conclude about pointer assignment concerning the data types?

**They must match no matter what. So I have to be careful.**

1. One of the reasons pointer variables are useful is that they provide an alternative means of accessing the memory location whose address is their value. To display or use the contents of the memory location that is contained in the pointer variable the dereferencing operator \* is used.

Add the statement to your program: **cout << \*intPtr2 << endl;**

This causes two actions to occur:

1. The contents of the memory location referred to by **intPtr2** is accessed. This is an address and
2. The contents of the memory location at that address is accessed.

Suppose we have the following code segment:

double \*ptr; // contains garbage, but is a pointer variable

double x = 5; // initialized to 5

Below is a graphical representation:

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name: | ptr |  | x |
| Contents: | garbage |  | 5 |

ptr = &x; // address of x is now stored in ptr.

Below is a graphical representation after the above statement executes:

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name: | ptr |  | x |
| Contents: | contains the address of x. E.g. 0x42FE |  | 5 |

\*ptr += 10; // Add ten to the contents of the memory location stored in the pointer variable.

Below is a graphical representation after the above statement executes:

|  |  |  |  |
| --- | --- | --- | --- |
| Variable name: | ptr |  | x |
| Contents: | contains the address of x. E.g. 0x42FE |  | 15 |

1. Now add the statement: **\*intPtr2 = 99;** to your program and then display the value of **int1**, **int2**, and **int3**. What changes has occurred? Explain in your own words how this was accomplished.

**Int2 changed to 99!**

1. Enter an output statement to display the values of:

**(intPtr2 – 1),** **intPtr2**, and (**intPtr2 + 1**).

What values appear?

**012FFC24**

**012FFC28**

**012FFC2C**

What relationship can you observe between these three values?

**They are 4 bytes apart, probably from adding and subtracting.**

Next modify your output statement to display the result of dereferencing these three values.

Explain what happened.

**It displayed the contents of the addresses except that it was added and subtracted based on what I put in.**

**Exercise 2**

1. Examine the following code.

double value = 29.7;

double \*ptr = &value;

1. Write a cout statement that uses the ptr variable to display the contents of the value variable.

Cout << \*ptr << endl;

1. Write an assignment statement that uses the ptr variable to add 13 to the contents of the value variable.

Cout << \*ptr + 13 << endl;

1. Each of the following deﬁnitions and program segments has errors. Rewrite them without errors.
   1. int x, \*ptr;

&x = ptr;

**int x, \*ptr;**

**ptr = &x;**

* 1. int x, \*ptr;

\*ptr = &x;

**Int x, \*ptr;**

**Ptr = &x;**

* 1. int x, \*ptr;

ptr = &x;

ptr = 100; // Store 100 in x

cout << x << endl;

**int x, \*ptr;**

**ptr = &x;**

**\*ptr = 100; // Store 100 in x**

**cout << x << endl;**

* 1. int numbers[] = {10, 20, 30, 40, 50};

cout << "The third element in the array is ";

cout << \*numbers + 3 << endl;

**int numbers[] = {10, 20, 30, 40, 50};**

**cout << "The third element in the array is ";**

**cout << \*(numbers + 2) << endl;**

* 1. int values[20], \*iptr;

iptr = values;

iptr \*= 2;

**int values[20], \*iptr;**

**iptr = values;**

**\*iptr \*= 2;**

* 1. float level;

int fptr = &level;

**float level;**

**int \*fptr = &level;**

1. Given the following declarations:

int \*p, \*q;

int r = 3, s = 4, t = 5;

Are the following operations valid or invalid? Why or why not? If valid, what is the result?

|  |  |
| --- | --- |
| a)  p = &r;  q = &r;  cout << (p == q) << endl;  **Yes, both pointers are initialized and the result is one.** | b)  \*p = r;  \*q = &s;  \*p = \*p + \*q;  cout << \*p << endl;  No, you can’t dereference an address. |
| c)  p = &t;  q = &r;  s = \*p + \* q;  cout << s << endl;  **Yes, s was simply the two values that p and q pointed to; eight.** | d)  p = &t;  q = &r;  s = p + q;  cout << s << endl;  **No, you can’t add two pointers.** |
| e)  p = &r;  p = 10;  cout << r << endl;  **No, you can’t change the address to 10.** | |

1. Do Lab 3.1 on pages 37-44.

**Due Dates:**

As indicated on the assignment three folder

**What to hand in:**

1. A print out of your word document.
2. Save your word document as yournameLab3.docx
3. Place the word document into the assignment folder lab experience three.

Lab 3.1

#include <iostream>

#include <iomanip>

using namespace std;

typedef int IntegerArray[16];

typedef char CharArray[10];

int main() {

IntegerArray first={ 0 };

IntegerArray arr={ 1 };

IntegerArray last={ 2 };

//IntegerArray prime{ 2,3,5,7,11,13,17,19,23,29,31,37,41,43, };

//for (int i = 0; i < 16; i++)

//cout << prime[i] << endl;

//CharArray animal{"zebra"};

//for (int i = 0; i < 10; i++)

//cout << strlen(animal) << endl;

arr[-10] = -999;

arr[20] = 999;

for (int i = 0; i < 15; i++)

cout << " " << first[i];

cout << endl;

for (int i = 0; i < 15; i++)

cout << " " << arr[i];

cout << endl;

for (int i = 0; i < 15; i++)

cout << " " << last[i];

cout << endl;

cout << "\narr[-10]..arr[20]:\n";

for (int i = -10; i <= 20; i++)

cout << arr[i] << " ";

cout << "\n\n\n\n\n\n\n";

int mat[3][4] = { {11,22,33,44},{55,66,77,88},{-1,-2,-3,-4} };

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 4; j++)

{

cout << setw(9) << mat[i][j];

}

cout << endl;

}

cout << \*(mat + 0) << endl;

cout << \*(mat + 1) << endl;

cout << \*(mat + 2) << endl;

cout << \*\*(mat + 0) << endl;

cout << \*\*(mat + 1) << endl;

cout << \*\*(mat + 2) << endl;

cout << \*(\*mat + 0) << endl;

cout << \*(\*mat + 1) << endl;

cout << \*(\*mat + 2) << endl;

cout << \*(\*(mat+1) + 0) << endl;

cout << \*(\*(mat + 1) + 1) << endl;

cout << \*(\*(mat + 1) + 2) << endl;

}

