**Lecture 16 - Chapter 7: Pointers – Mon Oct 16 or Tues Oct 17**

**Announcements**

Reading:

* Chapter 7

Assignments:

* Assign: Assignment #7 - due on **Oct 18** (MW class) or **Oct 19** (TR class) **(no late assignments accepted)**

Exam #2

* **Monday Oct 23** MW class, **Tuesday Oct 24** TR class (Chapters 5– 7)

**Today’s Goals**

1. Const Qualifier
2. Bubble Sort
3. Sizeof Operator
4. Pointer Expressions and Arithmetic
5. Midterm Review

**Today’s Terminology 5 min**

**Terminology**

* Pointer
  + Variable whose value is the address of another variable.
* Indirection
  + When you reference a value through a pointer
* Pass by Value
  + When a copy of an argument is sent to the function
* Pass by Reference
  + When the address of an argument is sent to a function
* Least Privilege Principle
  + Provide enough access to data but no more
* Function Pointer
  + Variable whose value is the address of a function!

**Const Qualifier**

**Const**

* Const was not available in earlier C versions – but now is and should be used!
* Use to tell complier to make variable a constant!
* Constants cannot be modified
* Complier will catch modification attempts (warning or error – depends on complier)
* Use when **need to** **pass by reference** but want **pass by value protection**!

Arrays Large structures (chapter 10)

Automatically Structures are pass by value but when large

passed by reference need to pass by reference for performance

* Think in terms of the principle of least privilege – if it doesn’t need access don’t give it
* In this example, we made parameter **array** a **const**
  + Linear search does not need to modify array to perform its task

size\_t **linearSearch** (**const** **int** array[], **int** key, size\_t size) {

. . .

}

* Four cases when passing pointer to function
  + **Case 1: Non-constant pointer to non-constant data**
    - Highest level of data access
      * Data can be modified through dereferenced pointer
      * Pointer can be modified to point to other data items
    - No use of **const**
    - Seen this with arrays

**int** **main**(void) {

**int** someValue = 100;

**int** \*someValuePtr = &someValue; // non-constant pointer

modifyTest(someValuePtr);

}

// Function can modify incoming pointer and the data pointed to

**void** **modifyTest** (**int** \*numPtr) {

Assume have print statements here

\*numPtr = \*numPtr +1; Changes 100 to 101

**int** newValue = 5;

numPtr = &newValue; I can change pointer value here with

no warning or error. Also, no effect in

main since address is passed by value

} // modifyTest Assume have print statements here

|  |  |  |
| --- | --- | --- |
|  | **Location** | **Value** |
|  |  |  |
| someValue | **0028FF10** | 100 |
|  |  |  |

**\*someValuePtr**

……..

This address is ……..

passed by value!

|  |  |  |
| --- | --- | --- |
| someValuePtr | 0028FF1C | **0028FF10** |
|  |  |  |
|  |  |  |

**Before call**

someValue before call = 100

someValuePtr before call = 0028FF10

\*someValuePtr before call = 100

**Inside modifyTest**

Before changes

\*numPtr = 100

numPtr = 0028FF10

After changes

\*numPtr = 101

numPtr = 0028FF2C

\*numPtr = 5

**After call**

someValue after call = 101

someValuePtr after call = 0028FF10 **Not affected since address**

\*someValuePtr after call = 101 **passed by value!**

* + **Case 2: Non-Constant pointer to constant data**
    - Data pointed to by pointer **cannot** be modified
    - Pointer can be modified

**int** **main**(void) {

**int** someValue = 100;

**int** \*somePtr = &someValue; // non-constant pointer

modifyTest(somePtr);

}

Read this right to left

numPtr is pointer to an integer constant

// Function CANNOT modify data that is pointed to by numPtr

**void** **modifyTest** (const **int** \*numPtr) {

Assume have print statements here

\*numPtr = \*numPtr +1; **Causes an error! Cannot do!**

**int** newValue = 5;

numPtr = &newValue; I can change pointer value here with

no warning or error. Also has no effect

in main since address passed by value

} // modifyTest Assume have print statements here

} // modifyTest

|  |  |  |
| --- | --- | --- |
|  | Location | Value |
|  |  |  |
| someValue | **0028FE10** | 100 |
|  |  |  |

**\*someValuePtr**

……..

This address is ……..

passed by value!

|  |  |  |
| --- | --- | --- |
| someValuePtr | 0028FF1C | **0028FF10** |
|  |  |  |
|  |  |  |

**Before call**

someValue before call = 100

someValuePtr before call = 0028FF10

\*someValuePtr before call = 100

**Inside modifyTest**

Before changes

\*numPtr = 100

numPtr = 0028FF10

After changes

~~\*numPtr = 101~~ Cannot modify because const

numPtr = 0028FE2C

\*numPtr = 5

**After call** Not modified since const

someValue after call = 100

someValuePtr after call = 0028FF10 Not affected since address

\*someValuePtr after call = 100 passed by value!

* + **Case 3: Constant pointer to a non-constant data**
    - Data pointed to by pointer can be modified
    - Pointer **cannot** be modified
      * Once it points to a memory address it will always point to that memory address
      * Default for array name – array name is a constant pointer to the beginning of array

**int** **main**(void) {

**int** someValue = 100; **Read from right to left**

**int** \* **const** someValuePtr = &someValue; **someValuePtr is a constant**

**pointer to an integer**

Assume have print statements here

\*somePtr2 = \*somePtr2 + 1;

**int** number = 5;

somePtr2 = &number; **Causes an error! Cannot do!**

Assume have print statements here

} // main

|  |  |  |
| --- | --- | --- |
|  | Location | Value |
|  |  |  |
| someValue | **0028FE10** | 100 |
|  |  |  |

**\*someValuePtr**

……..

This address is ……..

passed by value!

|  |  |  |
| --- | --- | --- |
| someValuePtr | 0028FF1C | **0028FF10** |
|  |  |  |
|  |  |  |

**Before changes**

someValue = 100

someValuePtr = 0028FF10

\*someValuePtr = 100

**After changes**

someValue = 101

someValuePtr = 0028FE10

\*someValuePtr = 101

* + **Case 4: Constant pointer to constant data**
    - Least level of data access
      * Data **cannot** be modified
      * Pointer **cannot** be modified

**int** **main**(void) {

someValue = 100;

**const** **int** \* **const** someValuePtr = &someValue; someValuePtr is a

constant pointer to

an integer constant

\*somePtr3 = \*somePtr3 + 1; **Causes an error! Cannot do!**

number = 5;

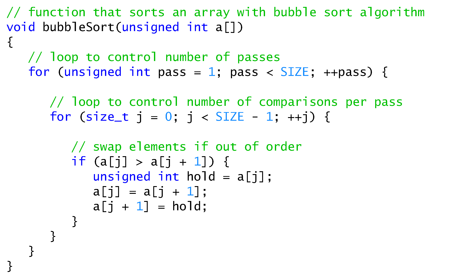
somePtr3 = &number; **Causes an error! Cannot do!**

}

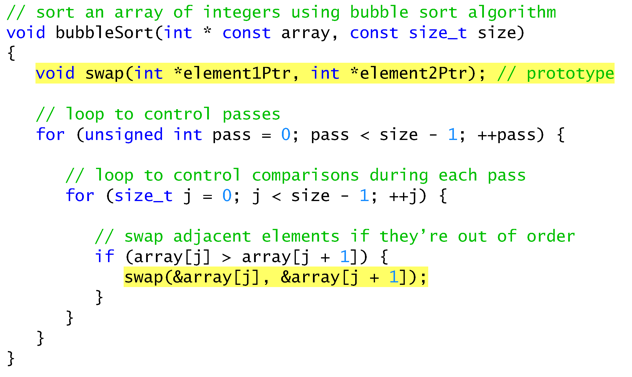
**Bubble Sort**

Bubble Sort

* Took code in bubble sort function and broke it up into two functions
* Original code
  + Declares that function receives a one-dimensional array
  + **int a[]**



* New code (figure 7.15)
  + Also declared function receives one-dimensional array
  + Remember **int a[]** is same as saying **int \*array**
  + **int \* const array –** including **const** qualifier so function cannot modify – enforce least privilege!



Notice the function

prototype for swap

is INSIDE bubbleSort

function!

bubbleSort is only

function that will use

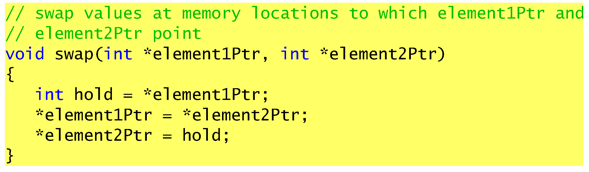
swap so place

prototype inside to

enforce least

privilege principle!

* + Swap is sent two pointers to array elements not the entire array
  + Swap has access to only the two elements since given address of each element
  + Remember:
    - An entire array is passed by reference
    - An individual element (int, float, etc.) in the array is passed by value
    - Must **explicitly** state you want array element passed by reference



**sizeof Operator 5 min**

**sizeof Operator**

* Use to determine the size in bytes of data types
* Compile time operator
* Returns value as a size\_t
* Values return by *sizeof* are implementation and platform dependent
* If checking the size of a variables, no parentheses are necessary

**int** number = 10;

**printf** ("sizeof an integer number = %u", **sizeof** number);

* If checking the size of a type, parentheses are necessary

**printf** ("sizeof an integer is = %u\n", **sizeof**(**int**));

**Examples**

**int** intNumber = 10;

**printf** ("sizeof an integer = %u\n", **sizeof**(intNumber));

**printf** ("sizeof an integer = %u\n", **sizeof**(**int**));

**puts**("");

**double** doubleNumber = 10.0;

**printf** ("sizeof a double = %u\n", **sizeof**(doubleNumber));

**printf** ("sizeof a double = %u\n", **sizeof**(**double**));

**puts**("");

**int** arrayOfTwentyIntegers[10];

**printf** ("sizeof an array of 10 integers = %u\n", **sizeof**(arrayOfTwentyIntegers));

**int** \*intNumberPtr = &number;

**printf** ("sizeof an integer pointer intNumberPtr = %u\n", **sizeof**(intNumberPtr));

**double** \*doubleNumberPtr = &doubleNumber;

**printf** ("sizeof a double pointer doubleNumberPtr = %u\n", **sizeof**(doubleNumberPtr));

**Displays**

Note: These are the values I got on my system, the values you see on your system could be different.

sizeof an integer = 4

sizeof an integer = 4

sizeof a double = 8

sizeof a double = 8

sizeof an array of 10 integers = 40

sizeof an integer pointer intNumberPtr = 4

sizeof a double pointer doubleNumberPtr = 4

**Pointer Expressions and Arithmetic 20 min**

**Pointer Expressions**

* Pointers can be used in these types of expressions
  + Arithmetic
  + Assignment
  + Comparison

**Pointing to an Array**

* Assume you have an array and a pointer variable

**int** v[5] = {100, 200, 300, 400, 500};

**int** \*vPtr;

* You can make vPtr point to the array in two ways:

vPtr = v;

vPtr = &v[0];

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0028FEE4 | 0028FEE8 | 0028FEEC | 0028FEF0 | 0028FEF4 |
| v[0] | v[1] | v[2] | v[3] | v[4] |
| 100 | 200 | 300 | 400 | 500 |

|  |
| --- |
|  |

pointer variable **vPtr**

vPtr = v; Showing there is no

**printf** ("Address of array v is = %p\n", v); difference if vPtr

**printf** ("vPtr = %p\n", vPtr); is assigned the

**printf** ("v[0] = %d\n", v[0]); value store in v or

**printf** ("\*vPtr = %d\n", \*vPtr); the address of v[0]

vPtr = &v[0];

**puts** ("");

**printf** ("Address of array &v[0] is = %p\n", &v[0]);

**printf** ("vPtr = %p\n", vPtr);

**printf** ("v[0] = %d\n", v[0]);

**printf** ("\*vPtr = %d\n", \*vPtr);

**Displays**

Address of array v is = **0028FEE4**

vPtr = **0028FEE4**

v[0] = 100

\*vPtr = 100

Address of array &v[0] is = **0028FEE4**

vPtr = **0028FEE4**

v[0] = 100

\*vPtr = 100

**Pointer Arithmetic**

* You can use the following operators on pointers
  + + (addition)
  + - (subtraction)
  + ++ (increment)
  + -- (decrement)
* Adding Integer
  + Pointer addition **depends** on the size of the pointer
  + Above, size of each integer is 4 bytes on my system, so

vPtr = vPtr + 2;

**printf** ("vPtr = %p\n", vPtr);

**printf** ("\*vPtr = %d\n", \*vPtr);

**Displays**

vPtr = 0028FEEC

\*vPtr = 300

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0028FEE4 | 0028FEE8 | 0028FEEC | 0028FEF0 | 0028FEF4 |
| v[0] | v[1] | v[2] | v[3] | v[4] |
| 100 | 200 | 300 | 400 | 500 |

|  |
| --- |
|  |

pointer variable **vPtr**

* Subtracting Integer

vPtr = vPtr - 2;

**printf** ("vPtr = %p\n", vPtr);

**printf** ("\*vPtr = %d\n", \*vPtr);

**Displays**

vPtr = 0028FEE4

\*vPtr = 100

* Incrementing and Decrementing

vPtr++; // Moves pointer to next location in array

vPtr--;

* Adding and Subtracting One Pointer from Another

**int** \*new1Ptr;

**int** \*new2Ptr;

new1Ptr = &v[0];

new2Ptr = &v[4];

vPtr = new1Ptr + new2Ptr; **Causes an error! Cannot do.**

**Pointer addition NOT allowed in C**

**int** numElements = new2Ptr - new1Ptr; But we can subtract pointers!

This is the difference between

the two array pointers in the

number of elements from new1Ptr

and new2Ptr

**printf** ("new2Ptr - new1Ptr = number of elements from new1Ptr to new2Ptr =

%d\n", numElements);

**Displays**

new2Ptr - new1Ptr = number of elements from new1Ptr to new2Ptr = 4

* Note:
  + Array elements are stored in contiguous memory
    - This is why you can add and subtract integer values to/from pointers to array elements
    - For non-array elements you cannot assume they’re in contiguous memory
      * You **cannot** add two pointers
      * You can subtract two pointers that refer to elements of the ***same*** array
  + Need to be careful when using pointers with arrays since you could run off the ends

// Array has only 5 locations so this will be off array into location after

// v[4] - one past array

vPtr = vPtr + 5;

**printf** ("Value of vPtr = %p\n", vPtr);

**printf** ("Value at vPtr = %d\n", \*vPtr);

**printf** ("Value at v[5] = %d\n", v[5]);

**Displays**

Value of vPtr = 0028FEF8

Value at \*vPtr = 10292856

Value at v[5] = 10292856

**Pointer Assignment**

* When assigning one pointer to another pointer
  + Both pointers must be of the **same type** or be a **generic pointer** (void \*ptr)
* Pointing to void
  + void pointers are generic pointers
  + void pointers can be assigned to pointer of any type
  + void ***cannot be dereferenced***
    - Contains the memory location for an unknown type
    - Compiler doesn’t know how many bytes are needed for value
    - Compiler cannot dereference without the number of bytes
    - Used to implement generic functions in C
* Example

**int** someValue = 10;

**int** \*intPtr;

**int** \*anotherIntPtr;

**void** \*voidPtr;

intPtr = &someValue;

anotherIntPtr = intPtr;

voidPtr = intPtr;

**printf** ("intPtr = %p\n", intPtr);

**printf** ("\*intPtr = %d\n", \*intPtr);

**printf** ("anotherIntPtr = %p\n", anotherIntPtr);

**printf** ("\*anotherIntPtr = %d\n", \*anotherIntPtr);

**printf** ("voidPtr = %p\n", voidPtr);

printf ("Value at voidPtr = %d\n", \*voidPtr); **Syntax error! Cannot**

**dereference void pointer**

**Displays**

intPtr = 0028FEDC

\*intPtr = 10

anotherIntPtr = 0028FEDC

\*anotherIntPtr = 10

voidPtr = 0028FEDC

**Comparing Pointers**

* Comparing pointers doesn’t make sense unless the pointers point to array elements of the **SAME** array
* The main comparison is checking if a pointer is equal to NULL
* You can use the following operators to compare pointers
  + Equality ( == !=)
  + Relational (< <= > >=)
  + Logical (&& || !)

**Exam #2 Review 15 min**

**Chapter 5 – C Functions**

* Understand the reason modular programming is important in programming
* Know how to use these 2 basic math functions - sqrt, pow
* Know how to define and create a function, so understand:
  + Function prototypes
  + Function invocation
  + Function definition
* Understand the following terms and their role in creating and using functions:
  + Return type
  + Formal parameters
  + Actual parameters
* Understand the difference between ***formal*** and ***actual*** parameters
* Understand the difference between a ***value-returning*** function and a ***void*** function
* Understand how to pass values to a function and how to get a value back
* Understand automatic conversions:
  + Know that conversions occur when argument types do not correspond to parameter types
  + Know what kinds of issues automatic conversions can cause
* Understand the scope of variables:
  + Know how to determine the scope of variables
  + What local variables are
  + Understand how the call stack works and how this relates to scope of variables
* Understand what ***pass by value*** means and the impact on variables when a function is called
* Know what recursion is on a high-level, you will NOT be asked to write a recursive function
* Know how to write a function if given a simple task
  + For example, write a function that computes the sum of the numbers from 1 to N
* Be able to explain code, trace code, and write code snippets with functions

**Chapter 6 – C Arrays**

* Know how to declare, create, and initialize an array (one or two dimensional)
* Know how to access elements within an array (one or two dimensional)
* Understand initialization of arrays
  + What is in array when you don’t initialize the array
  + Understand initializer lists (i.e. how they work, what if fewer items than array elements, etc.)
* How to manipulate elements within an array (i.e. compute sum, find largest, display elements, etc.)
* Understand what happens with arrays when
  + Code accesses elements in an array outside the array bounds
  + There is an off-by-one situation (mostly issue when forgetting arrays are zero-based)
* Understand character arrays
  + Know how to create and initialize strings using character arrays
  + Understand the importance of the null character in strings
  + How to read and display strings
* Understand and know how to ***pass arrays to functions***
  + Understand what ***pass by reference*** means and the impact when passing an array to a function
  + Know how to pass an entire array or a single element in the array to functions
  + Understand when to use ***const*** qualifier on an array in a parameter list
* Understand the bubble sort on a high level
  + What does it do
  + How efficient it is
* Know how to perform a linear search for a key value
* Understand the concept of how the binary search works on a high level
  + What does it do
  + How efficient it is
  + When it should or should not be used
* Know when one search is better over the other
* Be able to explain code, trace code, and write code snippets with arrays

**Chapter 7 – C Pointers**

* Understand what a pointer variable is
  + How pointers are different from other variables
  + What is the purpose of pointers
* Know how to create a pointer and initialize it
* Know how to use the pointer operators & and \*
  + Know what does each operator does
  + Know when do you use one over the other
  + Understand code that contains these operators
* Understand and know how to ***pass pointers to functions***
  + Understand what ***pass by reference*** means and the impact when passing a pointer to a function
  + Understand and know what needs to be in argument and parameter lists when passing pointers
* Understand the 4 different cases of using the **const** qualifier with pointers
  + Non-constant pointer to non-constant data
  + Constant pointer to non-constant data
  + Non-constant pointer to constant data
  + Constant pointer to constant data
* Know how to use the ***sizeof*** operator
* Know how to use pointers in expressions and how to perform pointer arithmetic
  + Comparing pointers
  + Incrementing and decrementing pointers
  + Adding and subtracting values from pointers
  + Subtracting one pointer from anther
* Understand what a void pointer is
* Understand the relationship between arrays and pointers
* Understand how to create, access, manipulate an array of pointers
* Understand pointers to functions at a high level
* Be able to explain code, trace code, and write code snippets with pointers

**Overall**

* Be prepared to write small snippets for:
* Value returning or void functions
  + Passing and returning scalar types (int, float, etc.)
  + Passing and returning arrays
  + Passing and returning pointers
* Manipulating arrays
  + One or two dimensional
* Manipulating pointers