**Lecture 17 - Chapter 7: Pointers – Wed Oct 18 or Thurs Oct 19**

**Announcements**

Reading:

* Chapter 8 – Starting next Mon

Assignments:

* Due: Assignment #6
* Assign: Assignment #7 - due on **Oct 25** (MW class) or **Oct 26** (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. Relationship Between Pointers and Arrays
2. Arrays of Pointers
3. Pointers to Functions
4. Secure C Programming

**Today’s Terminology**

**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!

**Relationships between Pointers and Arrays**

**Pointer and Arrays**

* Array name without index is pointer to 1st element in array

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

**int** \*vPtr;

// These two statements do same thing! Set vPtr to address of 1st element

vPtr = v;

vPtr = &v[0];

**Pointer/Offset Notation**

* Using pointers for array indexing

vPtr = v;

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

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

// v[3] and \*(vPtr + 3) refer to the same array element

**printf** ("v[3] = %d\n", v[3]);

**printf** ("\*(vPtr + 3) = %d\n", \*(vPtr + 3));

**Displays** Offset pointer by 3

v[0] = 100 Same as having pointer point to v[3],

\*vPtr = 100 in this case

v[3] = 400

\*(vPtr + 3) = 400

\*vPtr + 3 = 103 Shows what happens if leave of ()

* Basically, all indexed array expressions can be written with a pointer and an offset.

**Pointer/Index Notation**

* Pointers can we written using array [ ] notation!

**printf** ("vPtr[1] = %d\n", vPtr[1]);

**Displays**

vPtr[1] = 200

**Notes:**

* Array name
  + Points to first element in array
  + Cannot be modified otherwise we would lose connection to the array
  + Act like const pointer

**int** newArray[5] = {3, 5, 8, 1, 3};

**printf** ("newArray is address to first element - newArray = %p\n", newArray);

newArray = 0; **Causes an error! Cannot give array name a new value!**

**Arrays of Pointers**

**Purpose**

* Gives ability to create an array of where each element is a string
* In C, strings are basically character arrays
* Make each element in array is a pointer to a string

**Defining Array of Pointers**

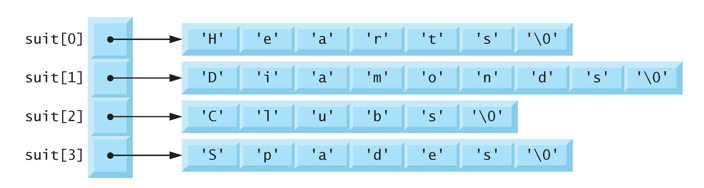
* In this example from the book it looks like an array with 4 strings is being created.
* Actually, an array with 4 pointers that point to strings is created.

**const** **char** \*suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};

Strings pointed to Each element in array Array contains 4 elements

by each element can is a *pointer to char* suit[0], suit[1], suit[2], suit[3]

not be modified



In this array, each pointer takes **4 bytes** of memory so needed 16 bytes to store above array

* Could use a 2-dimensional array but it would consume more memory
  + Each row is a suit – need 4 rows
  + Each column is a letter – need space for largest string + null = 9 columns to fit “Diamond”
  + Each character takes 1 byte of memory so need **36 bytes** to store this array

**char** suit[4][9] = {"Hearts", "Diamonds", "Clubs", "Spades"};

**printf** ("Suit[0] = %s\n", suit2[0]);

**printf** ("Suit[1] = %s\n", suit2[1]);

**printf** ("Suit[2] = %s\n", suit2[2]);

**printf** ("Suit[3] = %s\n", suit2[3]);

**printf** ("The size of suit is %d\n", **sizeof** suit);

**Displays**

Suit[0] = Hearts

Suit[1] = Diamonds

Suit[2] = Clubs

Suit[3] = Spades

The size of suit is 36

**Accessing Array of Pointer Elements**

* Accessing an array with pointer elements can be a bit confusing
* Here are a bunch of examples

**const** **char** \*suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};

|  |  |  |  |
| --- | --- | --- | --- |
| 0028FEBC | 0028FEC0 | 0028FEC4 | 0028FEC8 |
| suit[0] | suit[1] | suit[2] | suit[3] |
|  |  |  |  |
| 004086D6 | 004086DD | 004086E6 | 004086EC |
|  |  |  |  |
|  |  |  |  |
| ‘H’ | ‘D’ | ‘C’ | ‘S’ |
| ‘e’ | ‘i’ | ‘l’ | ‘p’ |
| ‘a’ | ‘a’ | ‘u’ | ‘a’ |
| ‘r’ | ‘m’ | ‘b’ | ‘d’ |
| ‘t’ | ‘o’ | ‘s’ | ‘e’ |
| ‘s’ | ‘n’ | ‘\0’ | ‘s’ |
| ‘\0’ | ‘d’ |  | ‘\0’ |
|  | ‘s’ |  |  |
|  | ‘\0’ |  |  |

// Address of each location in the array

**puts**("");

**printf** ("&suit[0] - address of 1st element = %p\n", &suit[0]);

**printf** ("&suit[1] - address of 2nd element = %p\n", &suit[1]);

**printf** ("&suit[2] - address of 3rd element = %p\n", &suit[2]);

**printf** ("&suit[3] - address of 4th element = %p\n", &suit[3]);

// Actual values in the array shown as pointers

**puts** ("");

**printf** ("suit[0] - value of 1st element as pointer = %p\n", suit[0]);

**printf** ("suit[1] - value of 2nd element as pointer = %p\n", suit[1]);

**printf** ("suit[2] - value of 3rd element as pointer = %p\n", suit[2]);

**printf** ("suit[3] - value of 4th element as pointer = %p\n", suit[3]);

// Actual values in the array shown as strings

**puts** ("");

**printf** ("suit[0] - value of 1st element as string = %s\n", suit[0]);

**printf** ("suit[1] - value of 2nd element as string = %s\n", suit[1]);

**printf** ("suit[2] - value of 3rd element as string = %s\n", suit[2]);

**printf** ("suit[3] - value of 4th element as string = %s\n", suit[3]);

// Dereferencing the pointer at each element

**puts** ("");

**printf** ("\*suit[0]- is the character = %c\n", \*suit[0]);

**printf** ("\*suit[1]- is the character = %c\n", \*suit[1]);

**printf** ("\*suit[2]- is the character = %c\n", \*suit[2]);

**printf** ("\*suit[3]- is the character = %c\n", \*suit[3]);

// Accessing the other characters in 1st string

**puts** ("");

**printf** ("suit[0][0] - is the character = %c\n", suit[0][0]);

**printf** ("suit[0][1] - is the character = %c\n", suit[0][1]);

**printf** ("suit[0][2] - is the character = %c\n", suit[0][2]);

**printf** ("suit[0][3] - is the character = %c\n", suit[0][3]);

**printf** ("suit[0][4] - is the character = %c\n", suit[0][4]);

**printf** ("suit[0][5] - is the character = %c\n", suit[0][5]);

**Displays**

&suit[0] - address of 1st element = 0028FEBC

&suit[1] - address of 2nd element = 0028FEC0

&suit[2] - address of 3rd element = 0028FEC4

&suit[3] - address of 4th element = 0028FEC8

suit[0] - value of 1st element as pointer = 004086D6

suit[1] - value of 2nd element as pointer = 004086DD

suit[2] - value of 3rd element as pointer = 004086E6

suit[3] - value of 4th element as pointer = 004086EC

suit[0] - value of 1st element as string = Hearts

suit[1] - value of 2nd element as string = Diamonds

suit[2] - value of 3rd element as string = Clubs

suit[3] - value of 4th element as string = Spades

\*suit[0]- is the character = H

\*suit[1]- is the character = D

\*suit[2]- is the character = C

\*suit[3]- is the character = S

suit[0][0] - is the character = H

suit[0][1] - is the character = e

suit[0][2] - is the character = a

suit[0][3] - is the character = r

suit[0][4] - is the character = t

suit[0][5] - is the character = s

**Pointers to Functions**

**What is Pointer to a Function**

* Arrays – name is **address** of 1st element
* Functions – name is **starting address** of the code for the function
* Function pointer is a variable whose value is the address of a function!
* This means, it stores the start of executable code.
* Pointer function variables can be
  + Passed to functions
  + Returned from functions
  + Stored in arrays
  + Assigned to other function pointers

**Purpose**

* Pointers to functions are useful in situations when you want to
  + Create a callback mechanism and need to pass address of function to another function
  + Store an *array of functions* to call dynamically
* Object oriented features in C++ are implemented using function pointers in C!
  + Virtual functions
  + Class methods

**Defining Pointers to Functions**

* Normal function definition

**void** **regularFunction** (**int** value) {

**printf** ("Made it to regularFunction and the value passed in was %d", value);

}

* Pointer to function

**void** (\*functionPtr)(**int**); // Define a pointer to a function

functionPtr = &regularfunction; // Assign pointer address of regularFunction

// Invoke regularFunction using the pointer functionPtr

// Dereference pointer to function to use the function

(\*functionPtr)(817);

**Displays**

Made it to function1 and the value passed in was 817

* Note
  + Could write the above code in one statement

**void** (\*functionPtr)(**int**) = &function1;

* + Be careful with the parentheses!
  + Forget parentheses - you then have declaration for a function that returns a void pointer

**void** \*functionPtr(**int**); // Not a pointer to a function!

// This is a function that returns a pointer!

* + Like arrays, the function name also serves as starting address to function
  + If use function name, can leave off the **&** operator and it works the same

**void** (\*functionPtr)(**int**) = regularFunction; // Removed &

functionPtr(817); // Removed \*

**Example**

* Function pointers useful in menu-driven programs

**#include** <stdio.h>

**#include** <stdlib.h>

**void** **deposit** (**float** amount);

**void** **withdraw** (**float** amount);

**void** **transfer** (**float** amount);

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

// Display menu of options user can select from

**puts** ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

**puts** (" Bank Options");

**puts** ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

**puts** ("1: Deposit");

**puts** ("2: Withdraw");

**puts** ("3: Transfer");

**puts** ("");

// Prompt user for menu option and amount

**int** menuOption;

**puts** ("Select an option from the menu");

**scanf**("%d", &menuOption);

**float** amount;

**puts** ("Select amount for transaction");

**scanf** ("%f", &amount);

// Initialize array of 3 pointers to functions that each take

// a float argument and return void

**void** (\*funcPtr[])(**float**) = {deposit, withdraw, transfer};

// Based on menu option selected, call correct function to do processing

**if** ((menuOption >= 1) && (menuOption <= 3)) {

(\*funcPtr[menuOption-1])(amount);

}

**else** {

**puts** ("Invalid menu option - bye");

}

} // main

**void** **deposit** (**float** amount) {

**printf** ("Deposit amount is $%.2f\n", amount);

} // deposit

**void** **withdraw** (**float** amount) {

**printf** ("Withdraw amount is $%.2f\n", amount);

} // withdraw

**void** **transfer** (**float** amount) {

**printf** ("Transfer amount is $%.2f\n", amount);

} // transfer

* Go over Figure 7.28 – array of function pointers

**Notes**

* Normal pointers point to data- function pointers point to code
* Function name can also be used to get functions’ address
* Can have array of regular pointers – can have array of pointers to functions
* Can pass pointer to function as argument to functions
* Can return pointer to function as return value from a function

**Secure C Programming**

**Secure Programming**

* To write code that uses techniques that can stand up to attacks
* This topic is an entire class so we won’t be focusing on this topic
* We will discuss some of the techniques

**CERT C Secure Coding Standard**

* CERT – Computer Emergency Response Team - [www.cert.org](http://www.cert.org)
* Publishes and promotes secure coding standards
* Standard for C
  + <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
* Standard for other lanaguages:
  + <https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

**Other CERT Guidelines Regarding Pointer**

* Misused pointers cause lots of issues and security vulnerabilities!
* Some guidelines for topics we discussed in chapter 7
  + Dereferencing NULL pointers generally cause programs t crash but can also allow attackers to execute code
  + Use const in function parameter lists when appropriate
    - On pointers
    - On data being pointed to