Chapter 1 - Introduction

Pointers and Memory

Why You Should Become Proficient with Pointers

Declaring Pointers

```
int num;

int *pi;
```

- whitespace around ‘*’ is unimportant

  num address 100 _____ pi address 104 _____ address 108 _____

- always initialize pointers before dereferencing

- a declared pointer of a type should eventually be assigned an address of that type

- uninitialized variables (including pointers) contain garbage

- pointers do not have built-in checks for the validity of their contents

- compilers often catch pointer errors

How to Read a Declaration

```
const int *pci;
```

- reading declarations backwards can sometimes help
- pci is a pointer variable to a constant integer

Address of Operator

```
num = 0;
pi = &num;
```

- & will return the operand’s address

- NOTE: pointers and integers are not the same even if they occupy the same number of bytes for a given architecture

- try to initialize a pointer as soon as possible

  ```
  int num;
  int *pi;
  pi = &num;
  ```

Displaying Pointer Values

```
printf("Address of num: %d Value: %d\n",&num, num);
printf("Address of pi: %d Value: %d\n",&pi, pi);
```

- %x - Displays the value as a hexadecimal number.
- %o - Displays the value as an octal number.
- %p - Displays the value in an implementation-specific manner; typically as a hexadecimal number.

Virtual memory and pointers

- For most modern architectures, each program assumes it has access to the computer’s entire physical memory so addresses displayed are virtual and the OS/kernel/MMU translate these to a real physical memory address when necessary

Dereferencing a Pointer Using the Indirection Operator

```
printf("%p\n",*pi); // Displays 5
*pi = 200;
```

```
    printf("%d\n",num); // Displays 200
```

- the dereference operator can be used for rvalues or lvalues
- when used with an lvalue, the target must be modifiable (i.e. non-const)

Pointers to Functions

```
    void (*foo)();
```

The Concept of Null

- the null concept - an abstraction to support the null pointer constant

- the null pointer constant - may or may not be a constant 0

- the NULL macro - often defined as ((void *) 0), i.e. a constant 0 cast to a void pointer

- the ASCII NUL - a byte containing all zeros, i.e. '\0'

- a null string - ''

- the null statement - a statement with a single semicolon

  ```
  ;
  ```

To NULL or not to NULL

- only use NULL in the context of pointers

Pointer to void

- a pointer to void has the same representation and alignment as a char pointer
- it is never equal to another pointer, however two void pointers set to NULL are equal

- only use void pointers for data pointers

Global and static pointers

- a global or static pointer is initialized to 0 on program start and are generally stored in the data segment of an executable

Pointer Size and Types

- most architectures keep pointer sizes the same regardless of the type
- generally range from 16 to 64 with 32 and 64 being the most common

Memory Models

- defines a size matrix for common integer/char/pointer types

Predefined Pointer-Related Types

- size_t - Created to provide a safe type for sizes
- ptrdiff_t - Created to handle pointer arithmetic
- intptr_t and uintprt_t - Used for storing pointer addresses

Understanding size_t

- represents max size of any built-in type in C as an unsigned int

- generally declared in stdio.h or stdlib.h

```
#ifndef __SIZE_T
#define __SIZE_T
typedef unsigned int size_t; #endif
```

- do not assume size_t is the same size as a pointer

- use %u or %lu to print size_t

Using the sizeof operator with pointers

- determine pointer size with the sizeof operator

Using intptr_t and uintptr_t

- used for storing pointers as addresses and are useful for converting pointers to their integer representation

  ```
  int num;
  intptr_t *pi = &num;
  ```

- uintptr_t usage requires a cast

  ```
  char c;
  uintptr_t *pc = (uintptr_t*)&c;
  ```

- use these in place of casting pointers to integers

Pointer Operators

Pointer Arithmetic

- Adding an integer to a pointer
  - results in adding the product of the integer times the number of bytes occupied by the underlying data type

    ```
    int vector[] = {28, 41, 7};
    int *pi = vector; // pi: 100
    printf("%d\n",*pi); // Displays 28
    ```

```
pi += 1; // pi: 104
printf("%d\n",*pi); // Displays 41
pi += 1; // pi: 108
printf("%d\n",*pi); // Displays 7
```

- Subtracting an integer from a pointer

```
int vector[] = {28, 41, 7};
int *pi = vector + 2; // pi: 108
printf("%d\n",*pi); // Displays 7
pi--; // pi: 104
printf("%d\n",*pi); // Displays 41
pi--; // pi: 100
printf("%d\n",*pi); // Displays 28
```

- Subtracting two pointers from each other
    - the difference between two pointers is the integer number of units by which they differ
- Comparing pointers
    - can use the standard comparison operators to compare pointers, i.e. <, >, <=, >=

Common Uses of Pointers

Multiple Levels of Indirection

- it is common to see double pointers or a pointer to a pointer, i.e. argv
- this simplifies certain operations

Constants and Pointers

- pointers to constants

```
const int limit = 500;
const int *pci; // pointer to a constant integer
pci = &limit;
```

- constant pointers to non-constants
    - pointer must be initialized to point to a non-constant variable
    - pointer cannot be modified
    - the variable pointed to can be modified

    ```
    int num;
    int *const cpi = &num;
    *cpi = limit;
    *cpi = 25;
    ```

- constant pointers to constants

```
const int * const cpci = &limit;
```

    - can be initialized with a non-constant variable
- pointer to constant pointer to constant

```
const int * const cpci = &limit;
const int * const * pcpci;
```

Pointer Type	**Pointer Modifiable**	**Data Pointed to Modifiable**
Pointer to a nonconstant	✓	✓
Pointer to a constant	✓	X
Constant pointer to a nonconstant	X	✓
Constant pointer to a constant	X	X

Chapter 2 - Dynamic Memory Management in C

- C programs do have a runtime system -> it includes heap management and the stack function call standard
- C99 introduced variable length arrays, i.e. the size of an array is determined at runtime rather than compile time. They still cannot change size once created.

Dynamic Memory Allocation

1. Use a malloc type function to allocate memory

2. Use this memory to support the application

3. Deallocate the memory using the free function

```
    int *pi = (int*) malloc(sizeof(int)); *pi = 5;
    printf("*pi: %d\n", *pi);
    free(pi);
```

- use free to deallocate memory
- do not access a pointer once the memory it points to has been freed
- generally assigning NULL to a freed pointer is a best practice
- the heap manager also stores additional information, including the block size
- do not access memory outside of the block

Memory Leaks

- happens when memory is allocated but not freed
    - when the memory's address is lost
    - the free function is never invoked
- memory leaks cause the amount of memory available to the heap manager to decrease

- extreme example of no free

```
    char *chunk; while (1) {
    chunk = (char*) malloc(1000000);
    printf("Allocating\n"); }
```

- losing the address

```
  int *pi = (int*) malloc(sizeof(int)); *pi = 5;
  ...
  pi = (int*) malloc(sizeof(int));
```

Dynamic Memory Allocation Functions

- malloc - Allocates memory from the heap
- realloc - Reallocates memory to a larger or smaller amount based on a previously allocated block of memory
- calloc - Allocates and zeros out memory from the heap
- free - Returns a block of memory to the heap

Using the malloc Function

- function signature

```
    void* malloc(size_t);
```

- typical use

```
    int *pi = (int*) malloc(sizeof(int));
```

- steps malloc performs

    1. Memory is allocated from the heap

2. The memory is not modified or otherwise cleared
3. The first byte’s address is returned

- malloc may return NULL - it is a good practice to check for NULL before use

```
int *pi = (int*) malloc(sizeof(int)); if(pi != NULL) {
// Pointer should be good
} else {
// Bad pointer
}
```

- before void pointers existed, explicit casts were required

- it is good practice to cast

  - They document the intention of the malloc function
  - They make the code compatible with C++ (or earlier C compiler), which require explicit casts

- however casts will cause a problem if no malloc header is included as C assumes functions return an int

- if a pointer is declared but not initialized/allocated, the pointer contains garbage

- use the sizeof operator with the integer multiple of the number of items for which to allocate space

```
double *pd = (double*)malloc(NUMBER_OF_DOUBLES * sizeof(double));
// NOT!
const int NUMBER_OF_DOUBLES = 10;
double *pd = (double*)malloc(NUMBER_OF_DOUBLES);
```

- no standard way to determine the size of the allocated space, some compilers do provide it though

- the max size available to the heap manager is system dependent

- cannot use a function call when initializing static or global variables, so cannot use malloc with them

- NOTE: from the compiler's perspective the initialization operator, =, and the assignment operator, =, are different

Using the calloc Function

- sets the memory to binary 0s

```
void *calloc(size_t numElements, size_t elementSize);
```

- can use malloc + memset

```
int *pi = malloc(5 * sizeof(int));
memset(pi, 0, 5* sizeof(int));
```

- calloc may have a memory overhead

Using the realloc Function

- use the realloc function to resize memory
- First Parameter Second Parameter Behavior
- null NA Same as malloc
- Not null 0 Original block is freed
- Not null Less than the original block's size A smaller block is allocated using the current block
- Not null Larger than the original block's size A larger block is allocated either from the current location or another region of the heap

```
        char *string1;
        char *string2;
        string1 = (char*) malloc(16); strcpy(string1, "0123456789AB");
        string2 = realloc(string1, 8);
        printf("string1 Value: %p [%s]\n", string1, string1);
        printf("string2 Value: %p [%s]\n", string2, string2);
```

The alloca Function and Variable Length Arrays

- the alloca function is Microsoft's malloca, is stack based on non-portable

Deallocating Memory Using the free Function

- use free with pointers from malloc type functions

```
        int *pi = (int*) malloc(sizeof(int)); ...
        free(pi);
```

- Manage memory allocation/deallocation at the same level. For example, if a pointer is allocated within a function, deallocate it in the same function.

Assigning NULL to a Freed Pointer

- assigning NULL to a freed pointer can help catch issues by causing a runtime exception if a freed pointer is used
- it is best to search problems at their root rather than using this as a catch all
- additionally, NULL cannot be assigned to a const pointer when it is initialized

Double Free

- double frees result in a runtime exception

The Heap and System Memory

- free does not necessarily give system memory to the operating system, it generally just frees memory back to the application

Freeing Memory upon Program Termination

- making extra effort to free all memory before program termination
    - May be more trouble than it's worth
    - Can be time consuming and complicated for the deallocation of complex structures
    - Can add to the application's size
    - Results in longer running time
    - Introduces the opportunity for more programming errors

Dangling Pointers

- a pointer that still references memory after the memory has been freed, aka premature free
- results in:
    - Unpredictable behavior if the memory is accessed
    - Segmentation faults when the memory is no longer accessible
    - Potential security risks
- happen when:
    - Memory is accessed after it has been freed
    - A ointer is returned to an automatic variable in a previous function call

Dangling Pointer Examples

```
        int *pi = (int*) malloc(sizeof(int)); *pi = 5;
        printf("*pi: %d\n", *pi);
        free(pi);
```

- pi still holds the integer's address

```
      int *p1 = (int*) malloc(sizeof(int)); *p1 = 5;
      ...
      int *p2;
      p2 = p1;
      ...
      free(p1);
      ...
      *p2 = 10;
```

- p2 still references memory freed by p1

```
    int *pi;
    ...
    {
        int tmp = 5;
        pi = &tmp;
    }
    // pi is now a dangling pointer
    foo();
```

- most compilers treat blocks as a stack frame and pi points to an address that will probably be overridden by a stack variable

Dealing with Dangling Pointers

- Setting a pointer to NULL after freeing it. Its subsequent use will terminate the application. However, problems can still persist if multiple copies of the pointer exist. This is because the assignment will only affect one of the copies.
- Writing special functions to replace the free function.
- Some systems (runtime/debugger) will overwrite data when it is freed. While no exceptions are thrown, when the programmer sees memory containing these values where they are not expected, he/she knows that the program may be accessing freed memory.
- Use third-party tools to detect dangling pointers and other problems.

Debug Version Support for Detecting Memory Leaks

- most compilers include special memory management techniques in debug builds to:
    - Check the heap's integrity
    - Check for memory leaks
    - Simulate low heap memory situations

Dynamic Memory Allocation Technologies

- most heap managers use heap or data segment as the source for memory
- solves many problems like if heap is per process or per thread and security
- many flavors

Garbage Collection in C

- garbage collection provides the following:
    - Freeing the programmer from having to decide when to deallocate memory
    - Allowing the programmer to focus on the application's problem

Resource Acquisition Is Initialization

- designed for C++
- many approaches to mirror in C, with GNU support for it
- GNU provides a macro, RAII_VARIABLE, that associates a type, a function to execute when the variable is created and a function to execute when the variable goes out of scope with a variable
- see the GCC cleanup extension for C

Using Exception Handlers

```
    void exceptionExample() { int *pi = NULL;
    __try {
        pi = (int*)malloc(sizeof(int)); *pi = 5;
        printf("%d\n",*pi);
    }
```

```
        __finally {
            free(pi); }
        }
```

- Exception handling can be added to C using a number of approaches

Chapter 3 - Pointers and Functions

Program Stack and Heap

- important runtime elements
- stack variables are called automatic variables and are allocated to stack frames

Program Stack

- stack is an area of memory that supports function execution
- function calls push their frames onto the stack and the stack grows upwards
- dynamic memory tends to grow downwards

Organization of a Stack Frame

- return address - the address in the program where the function is to return upon completion
- storage for local data - emory allocated for local variables
- storage for parameters - memory allocated for the function's parameters
- stack and base pointers - pointers used by the runtime system to manage the stack
- NOTE: this describes the C calling convention
- C treats blocks similarly to functions and pushes/pops as necessary

Passing and Returning by Pointer

- allows an object to be accessible in multiple functions without making it global
- if modified within the function, an object needs to be passed by pointer
- pass by pointer to const to prohibit modification (and avoid pass-by-value copy overhead)

Passing Data Using a Pointer

- the function can modify the data

```
void swapWithPointers(int* pnum1, int* pnum2) { int tmp;
    tmp = *pnum1;
    *pnum1 = *pnum2;
    *pnum2 = tmp;
}
```

Passing Data by Value

- the previous example is not possible with pass by value because the parameters are copies

Passing a Pointer to a Constant

- provides efficiency but avoids the potential for modification

```
void passingAddressOfConstants(const int* num1, int* num2) {
    *num2 = *num1;
}
```

Returning a Pointer

- declare function to return a pointer to type
- use one of the following two techniques:
    1. Allocate memory within the function using malloc and return its address. The caller is responsible for deallocating the memory returned.
    2. Pass an object to the function where it is modified. This makes the allocation and deallocation of the object's memory the caller's responsibility.
- problems that can occur:
    - Returning an uninitialized pointer

    - Returning a pointer to an invalid address
    - Returning a pointer to a local variable
    - Returning a pointer but failing to free it

Pointers to Local Data

- do not return pointers to local data as the address will point to a stack frame that will probably be/have been modified

Passing Null Pointers

- it is good practice to check for null pointers within functions that accept pointers as arguments

Passing a Pointer to a Pointer

- pointers are passed by values

- if allocating space for an array, pass a pointer to a pointer

- writing your own free function:

```
void saferFree(void **pp) {
    if (pp != NULL && *pp != NULL) {
        free(*pp);
        *pp = NULL;
    }
}

#define safeFree(p) saferFree((void**)&(p))
```

Function Pointers

- a pointer that holds the address of a function
- risks a potentially slower running program

- function pointer usage in certain situations can mitigate this

Declaring Function Pointers

- function pointers require both a return type and argument types which constitute something akin to a function pointer signature

- function pointers require that the programmer checks their use as the compiler does not necessarily check

```
    int (*f1)(double);
    void (*f2)(char*);
    double* (*f3)(int, int);
```

- some conventions require that a function pointer name is always prefixed with fptr

Using a Function Pointer

- to use a function pointer, assign a function with the same signature to the pointer

```
    fptr1 = square; // or
    fptr1 = &square;
```

- memory for the function pointer is allocated within the stack frame within which it is created and is the size of a pointer. It generally points to a segment within the text/code segment of an executable

Passing Function Pointers

- function pointers can be passed using their signature or using typedefs

```
    typedef int (*fptrOperation)(int,int);
    int compute(fptrOperation operation, int num1, int num2) {
    // or
    int compute(int (*operation)(int, int), int num1, int num2) {
```

Returning Function Pointers

- returning function pointers requires declaring the return type using a typedef that is a function pointer signature

Using an Array of Function Pointers

- can declare arrays of function pointers with a typedef or using the signature

```
    typedef int (*operation)(int, int);
    operation operations[128] = {NULL};
    // or
    int (*operations[128])(int, int) = {NULL};
```

- NOTE: can use the implicit conversion of chars to ints to use chars as array indices

Comparing Function Pointers

- can use '==' to check for equality of function pointer signatures

Casting Function Pointers

- function pointers can be cast between signatures but should be done with care as it is not guaranteed to work

```
        typedef int (*fptrToSingleInt)(int);
        typedef int (*fptrToTwoInts)(int,int);
        int add(int, int);
        fptrToTwoInts fptrFirst = add;
        fptrToSingleInt fptrSecond = (fptrToSingleInt)fptrFirst;
        fptrFirst = (fptrToTwoInts)fptrSecond; printf("%d\n",fptrFirst(5,6));
```

- void * is not guaranteed to work with function pointers

- always use the correct argument list with function pointers

Chapter 4 - Pointers and Arrays

- array names and pointers can be used interchangeably in many contexts but an array name is not a pointer and cannot always be used in place of each other
- an array name will return an address but the array name by itself cannot be used as the target of an assignment

Quick Review of Arrays

- array - contiguous collection of homogeneous elements that can be accessed using indices
- arrays have a fixed size but realloc and variable length arrays provide a means to resize based on size needs

One-Dimensional Arrays

- declaration:

```
int vector[5];
```

- initialization:

```
int vector[5] = {1, 2, 3, 4, 5};
```

- obtaining the size:

```
sizeof(vector)/sizeof(int);
```

Two-Dimensional Arrays

- declaration/initialization:

```
      int matrix[2][3] = {{1,2,3},{4,5,6}};
```

- memory layout:

```
  - matrix[0][0] 100 - 1
  - matrix[0][1] 100 - 2
  - matrix[0][2] 100 - 3
  - matrix[1][0] 100 - 4
  - matrix[1][1] 100 - 5
  - matrix[1][2] 100 - 6
```

- conceptual layout:

```
  column  0   1   1
  row
  0       1   2   3
  1       4   5   6
```

- access:

```
      for (int i = 0; i < 2; i++) {
          printf("&matrix[%d]: %p sizeof(matrix[%d]): %d\n",
              i, &matrix[i], i, sizeof(matrix[i]));
      }
```

Multidimensional Arrays

- declaration/initialization:

```
      int arr3d[3][2][4] = {
          {{1, 2, 3, 4}, {5, 6, 7, 8}},
          {{9, 10, 11, 12}, {13, 14, 15, 16}}, {{17, 18, 19, 20}, {21, 22, 2
3, 24}}
      };
```

Pointer Notation and Arrays

- when an array name is used by itself, it returns an address which can be assigned to a pointer

```
      int vector[5] = {1, 2, 3, 4, 5};
      int *pv = vector;
```

- the address of the array's first element also returns the same address

```
      printf("%p\n",vector);
      printf("%p\n",&vector[0]);
```

- each approach returns a pointer to an integer whereas &vector returns the address of the entire array
- array elements can be accessed in two ways

```
      pv[i]
      *(pv + i)
```

- array notation is like "shift and dereference"

Differences Between Arrays and Pointers

- vector[i] and vector + i
    - the machine code is different. vector[i] moves i positions from the location pointed to by 'vector' and uses the contents of the new location. vector + i uses the address as a value, adds i to the

    address and then gets the contents of that address
  - the sizeof operator will return the number of elements times the size of the element type when used with an array whereas it will return the size of a pointer when used with a pointer
  - a pointer is an lvalue and as such must be modifiable. An array name is not modifiable.

Using malloc to Create a One-Dimensional Array

- the memory sits on the heap and is the size of the content type times the size of the array

```
int *pv = (int*) malloc(5 * sizeof(int));
for(int i=0; i<5; i++) {
    pv[i] = i+1;
}

for(int i=0; i<5; i++) {
    *(pv+i) = i+1;
}
```

- NOTE: use *(pv + i) over *pv + i as the dereference operator has a higher precedence

Using the realloc Function to Resize an Array

- an array created with malloc can be resized with realloc

- realloc needs to be used for everything before C99 to resize arrays

- using realloc to handle user input:

```
char* getLine(void) {
    const size_t sizeIncrement = 10; char* buffer = malloc(sizeIncremen
t); char* currentPosition = buffer; size_t maximumLength = sizeIncrement; size
_t length = 0;
    int character;
```

```
            if(currentPosition == NULL) { return NULL; }
            while(1) {
            character = fgetc(stdin); if(character == '\n') { break; }
            if(++length >= maximumLength) {
            char *newBuffer = realloc(buffer, maximumLength += sizeIncrement);
            if(newBuffer == NULL) {
                free(buffer);
                return NULL;
            }
                            currentPosition = newBuffer + (currentPosition - buff
er);
                            buffer = newBuffer;
                        }
                        *currentPosition++ = character;
                    }
                    *currentPosition = '\0';
            return buffer;
        }
```

- realloc can also be used to decrease the amount of space used by a pointer

Passing a One-Dimensional Array

- when a one-dimensional array is passed to a function, the array's address is passed by value
- normally, this means that the size should also be passed because the function only knows the array's address
- in the case of a string stored in an array, we can rely on the '\0' character to find the end of the string
- array notation and pointer notation can be used

Using Array Notation

```
        void displayArray(int arr[], int size) {
            for (int i = 0; i < size; i++) {
                printf("%d\n", arr[i]);
        }
    }
```

- it is common practice to pass a smaller number than the full length of the array to only process a portion of the array if the array is not full of data

Using Pointer Notation

```
    void displayArray(int* arr, int size) {
        for (int i = 0; i < size; i++) {
            printf("%d\n", arr[i]);
        }
    }
```

- array notation or pointer notation can be used in the function body

```
    printf("%d\n", *(arr+i));
```

- pointer notation can also be used in the function body even if array notation is used in the declaration

Using a One-Dimensional Array of Pointers

- declares an array of integer pointers, allocates memory for an int for each element, and initializes the memory to the array's index

```
    int* arr[5];
    for(int i=0; i<5; i++) {
        arr[i] = (int*)malloc(sizeof(int));
        *arr[i] = i;
    }
```

- can also use pointer notation in the loop body

```
*(arr+i) = (int*)malloc(sizeof(int));
**(arr+i) = i;
```

- expressions

```
*arr[0] 0
**arr 0
**(arr+1) 1
arr[0][0] 0
arr[3][0] 3
```

Pointers and Multidimensional Arrays

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

- pointer for use with this matrix

```
int (*pmatrix)[5] = matrix;
```

- interpretation of a two-dimensional array

```
  arr[i][j]
  address of arr + (i * size of row) + (j * size of row)
```

Passing a Multidimensional Array

- always specify the array's shape. Use either:

```
void display2DArray(int arr[][5], int rows) {
void display2DArray(int (*arr)[5], int rows) {
```

- the following declaration does not work properly:

  ```
  void display2DArray(int *arr[5], int rows) {
  ```

  the array passed is assumed to be a five element array of pointers to integers

- if passed this way:

  ```
  void display2DArrayUnknownSize(int *arr, int rows, int cols) {
  ```

  it can be invoked this way:

  ```
  display2DArrayUnknownSize(&matrix[0][0], 2, 5);
  ```

  but array subscripts cannot be used, as in:

  ```
  printf("%d ", arr[i][j]);
  ```

  it must be accessed in one of the two following ways:

  ```
  printf("%d ", *(arr + (i*cols) + j));
  // or
  printf("%d ", (arr+i)[j]);
  ```

Dynamically Allocating a Two-Dimensional Array

- consider whether the array needs to be contiguous or if the array needs to be Jagged

- when declared:

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

memory is allocated contiguously

- when allocated with malloc, the “inner” elements of the arrays may not be contiguous
- if the memory is not contigous, copying the memory may require multiple copy operations

Allocating Potentially Non-contiguous Memory

```
    int rows = 2;
    int columns = 5;
    int **matrix = (int **) malloc(rows * sizeof(int *));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int *) malloc(columns * sizeof(int));
    }
```

Allocating Contiguous Memory

- first approach:

  ```
      int rows = 2;
      int columns = 5;
      int **matrix = (int **) malloc(rows * sizeof(int *));
      matrix[0] = (int *) malloc(rows * columns * sizeof(int));
      for (int i = 1; i < rows; i++)
          matrix[i] = matrix[0] + i * columns;
  ```

- second approach:

  ```
      int *matrix = (int *)malloc(rows * columns * sizeof(int));
  ```

  with this approach, array subscripts cannot be used:

```
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            *(matrix + (i*columns) + j) = i*j;
        }
    }
```

Jagged Arrays and Pointers

- a compound literal is a C construct that consists of what appears to be a cast operator followed by a an initializer list enclosed in braces:

```
    (const int) {100}
    (int[3]) {10, 20, 30}
```

- 3 x 3 array:

```
    int (*(arr1[])) = {
        (int[]) {0, 1, 2},
        (int[]) {3, 4, 5},
        (int[]) {6, 7, 8}
    };
```

- a jagged array:

```
    int (*(arr2[])) = {
        (int[]) {0, 1, 2, 3},
        (int[]) {4, 5},
        (int[]) {6, 7, 8}
    };
```

Chapter - 5 Pointers and Strings

String Fundamentals

- a string is a sequence of characters terminated by '\0' and are often represented as char *
- strings are commonly stored in arrays
- not all arrays of char are strings as not all arrays of char contain '\0'
- sometimes arrays of char are used to represent smaller integer units (e.g. boolean)
- two types of strings: char and wchar_t
- the length of a string is the number of characters in a string excluding '\0'
- remember to always allocate enough space for the '\0' character, i.e. add 1 to array sizes and malloc one additional for dynamic memory

String Declaration

- string literal, char array, char pointer

```
    char header[32];
    char *header;
```

The String Literal Pool

- string literals are often assigned to a literal pool, i.e. an area of memory to store immutable string literals
- this means multiple uses of a literal reference only one copy
- not good practice to assume there is only one copy or that a string is immutable as most compilers offer an option to turn off string literal pooling, i.e. -fwritable-strings for GCC or /GF for Visual Studio
- uses read-only memory
- often, it is good practice to declare strings as const char * or const char[] to avoid the ability to modify the string

String Initialization

- array of char:

```
    char header[] = "Media Player";
    // or
    char header[13];
    strcpy(header,"Media Player");
```

- characters can also be assigned individually

- pointer to char:

```
    char *header = (char*) malloc(strlen("Media Player")+1);
    strcpy(header,"Media Player");
    // or
    char *header = (char*) malloc(13);
    strcpy(header,"Media Player");
```

- when declaring the length of a string
  - always add one for '\0'
  - don't use sizeof, use strlen (or strlen_s if available)
- can also assign chars individually with char * and pointer notation access

- cannot initialize a char * with a character literal

- string placement in memory:

```
    char* globalHeader = "Chapter"; // globalHeader var is in global/static memory and "Chapter" is in the string literal pool
    char globalArrayHeader[] = "Chapter"; // globalArrayHeader var and and "Chapter" string are in global/static memory
    void displayHeader() {
        static char* staticHeader = "Chapter"; // staticHeader var is in global/static memory and "Chapter" is in the string literal pool
        char* localHeader = "Chapter"; // localHeader var is on the stack a
```

```
nd "Chapter" is in the string literal pool
          static char staticArrayHeader[] = "Chapter"; // staticArrayHeader v
ar and and "Chapter" string are in global/static memory
          char localArrayHeader[] = "Chapter"; // localArrayHeader var and "C
hapter" string are on the stack
          char* heapHeader = (char*)malloc(strlen("Chapter")+1); // heapHeade
r var is on the stack and points to "Chapter" string on the heap
          strcpy(heapHeader,"Chapter");
      }
```

Standard String Operations

Comparing Strings

- strcmp (do not use ‘=’ or ” - ” compares the address of the variable with the address of the string literal)

Copying Strings

- strcpy
    - the basic approach for reading strings in is read the string into a large array, malloc the right amount of memory and use strcpy to to copy the string into the malloc’d memory

Concatenating Strings

- strcat

Passing Strings

Passing a Simple String

```
        size_t stringLength(char* string) {
        // or
        size_t stringLength(char string[]) {
            size_t length = 0;
            while(*(string++)) {
                length++;
            }
            return length;
        }
```

Passing a Pointer to a Constant char

- better because it avoids the potential for the modification of the string

  ```
  size_t stringLength(const char* string) {
  ```

Passing a String to Be Initialized

- when using a buffer:
  - the address and the size must be passed
  - the caller is responsible for deallocating the buffer
  - the function normally returns a pointer to the buffer

    ```
    char* format(char *buffer, size_t size,
    ...
    return buffer;
    ```

- can handle if buffer address is NULL and then allocate the size needed

Passing Arguments to an Application

```
    int main(int argc, char** argv) {
    int main(int argc, char* argv[]) {
```

Returning Strings

Returning the Address of a Literal

- the address of a literal can be returned if literal pooling is used

```
char* returnALiteral(int code) {
...
return "Boston Processing Center";
```

- can also return the address of declared static literals

```
char* returnAStaticLiteral(int code, int subCode) {
    static char* bpCenter = "Boston Processing Center";
    ...
    return bpCenter;
```

this can present problems though

Returning the Address of Dynamically Allocated Memory

```
char* blanks(int number) {
    char* spaces = (char*) malloc(number + 1);
    ...
    return spaces;
```

- remember to assign '\0' to the last element
- NOTE: returning the address of a local string will cause problems and should not be done (the address of the stack frame that points to the local string will eventually be overwritten)

Function Pointers and Strings

- a function pointer can be passed as a comparator to sort strings

```
typedef int (fptrOperation)(const char*, const char*);
void sort(char *array[], int size, fptrOperation operation) {
    int swap = 1;
    while(swap) {
        swap = 0;
        for(int i=0; i<size-1; i++) {
            if (operation(array[i],array[i+1]) > 0) {
                swap = 1;
                char *tmp = array[i];
                array[i] = array[i+1];
                array[i+1] = tmp;
            }
        }
    }
}
```

Chapter 6 - Pointers and Structures

Introduction

- simple structure declaration:

```
struct _person {
    char* firstName;
    char* lastName;
    char* title;
    unsigned int age;
};
```

- use with typedef:

```
typedef struct _person {
    char* firstName;
    char* lastName;
    char* title;
    unsigned int age;
} Person;
```

- use with typedef:

```
Person person;
```

- use without typedef:

```
struct _person person;
```

- dynamic memory allocation:

```
Person *ptrPerson;
ptrPerson = (Person*) malloc(sizeof(Person));
```

- simple variable access:

```
person.firstName ...
```

- pointer variable access:

```
ptrPerson->firstName ...
```

How Memory Is Allocated for a Structure

- the amount of memory allocated is at minimum the sum of the size of the fields

- it is sometimes larger because padding can occur as a result of the need for the alignment of certain types of data on certain boundaries
- As a result:
    - pointer arithmetic must be used with care
    - arrays of structures may have extra memory between their elements

Structure Deallocation Issues

- a structure containing pointers will not automatically deallocate the pointers when the structure is deallocated
- it is good practice to define initialization and deallocation functions that take pointers to structure types to manage this

Avoiding malloc/free Overhead

- dynamic allocation of structures can incur significant overhead
- a good practice to avoid this is to keep a structure pool
    - create an instance if no instances are available
    - use an available instance otherwise
    - return instances to the pool when they are no longer necessary
- use more sophisticated list management schemes if the list size stays too small or grows too large

Using Pointers to Support Data Structures

Single-Linked List

```
        typedef struct _node {
            void *data;
            struct _node *next;
        } Node;

        typedef struct _linkedList {
            Node *head;
            Node *tail;
            Node *current;
```

```
} LinkedList;

void initializeList(LinkedList *list) {
    list->head = NULL;
    list->tail = NULL;
    list->current = NULL;
}

void addHead(LinkedList *list, void* data) {
    Node *node = (Node*) malloc(sizeof(Node));
    node->data = data;
    if (list->head == NULL) {
        list->tail = node;
        node->next = NULL;
    } else {
        node->next = list->head;
    }
    list->head = node;
}

void delete(LinkedList *list, Node *node) {
    if (node == list->head) {
        if (list->head->next == NULL) {
            list->head = list->tail = NULL;
        } else {
            list->head = list->head->next;
        }
    } else {
        Node *tmp = list->head;
        while (tmp != NULL && tmp->next != node) {
            tmp = tmp->next;
        }
        if (tmp != NULL) {
            tmp->next = node->next;
        }
        }
        free(node);
    }
```

Using Pointers to Support a Queue

```
    typedef LinkedList Queue;
```

- the queue needs an additional function to remove a tail node, i.e. dequeue:
    - An empty queue - NULL is returned
    - A single node queue - Handled by the else if statement
    - A multiple node queue - Handled by the else clause

    ```
    void *dequeue(Queue *queue) {
        Node *tmp = queue->head; void *data;
        if (queue->head == NULL) {
            data = NULL;
        } else if (queue->head == queue->tail) {
            queue->head = queue->tail = NULL;
            data = tmp->data;
            free(tmp);
        } else {
            while (tmp->next != queue->tail) {
                tmp = tmp->next;
            }
            queue->tail = tmp;
            tmp = tmp->next;
            queue->tail->next = NULL;
            data = tmp->data;
            free(tmp);
        }
    ```

```
        return data;
    }
```

Using Pointers to Support a Stack

```
typedef LinkedList Stack;
```

- uses push and pop

```
void *pop(Stack *stack) {
    Node *node = stack->head;
    if (node == NULL) {
        return NULL;
    } else if (node == stack->tail) {
        stack->head = stack->tail = NULL;
        void *data = node->data;
        free(node);
        return data;
    } else {
        stack->head = stack->head->next;
        void *data = node->data;
        free(node);
        return data;
    }
}
```

Using Pointers to Support a Tree

```
typedef struct _tree {
    void *data;
    struct _tree *left;
    struct _tree *right;
} TreeNode;
```

```
void insertNode(TreeNode **root, COMPARE compare, void* data) {
    TreeNode *node = (TreeNode*) malloc(sizeof(TreeNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    if (*root == NULL) {
        *root = node;
        return;
    }

    while (1) {
        if (compare((*root)->data, data) > 0) {
            if ((*root)->left != NULL) {
                *root = (*root)->left;
            } else {
                (*root)->left = node; break;
            }
        } else {
            if ((*root)->right != NULL) {
                *root = (*root)->right;
            } else {
                (*root)->right = node; break;
            }
        }
    }
}

void inOrder(TreeNode *root, DISPLAY display) {
   if (root != NULL) {
        inOrder(root->left, display);
        display(root->data);
        inOrder(root->right, display);
    }
```

```
        }

        void postOrder(TreeNode *root, DISPLAY display) {
            if (root != NULL) {
                postOrder(root->left, display);
                postOrder(root->right, display);
                display(root->data);
            }
        }

        void preOrder(TreeNode *root, DISPLAY display) {
            if (root != NULL) {
                display(root->data);
                preOrder(root->left, display);
                preOrder(root->right, display);
            }
        }
```

Chapter 7 - Security Issues and the Improper Use of Pointers

- use the CERT website and docs for more information

Pointer Declaration and Initialization

Improper Pointer Declaration

- declare pointers correctly:

  ```
  int *ptr1, *ptr2;
  // not
  int* ptr1, ptr2;
  ```

- prefer declaring vars on one line per var to this approach

- prefer typedefs to macros - facilitates compiler checks and debugging

Failure to Initialize a Pointer Before It Is Used

- do not use a pointer before it is initialized (wild pointer)

Dealing with Uninitialized Pointers

- cannot examine pointers to determine validity
- three approaches are used to check pointers:
    - always initialize a pointer to NULL

    ```
    int *pi = NULL; ...
    if(pi == NULL) {
              // pi should not be dereferenced
    } else {
    // pi can be used
    }
    ```

    - use the assert function

    ```
    assert(pi != NULL);
    ```

    - use third-party tools

Pointer Usage Issues

- buffer overflow presents a number of issues and it makes it possible to overwrite the return address of the stack frame with a call to malicious code
    - Not checking the index values used when accessing an array's elements
    - Not being careful when performing pointer arithmetic with array pointers

    - Using functions such as gets to read in a string from standard input
    - Using functions such as strcpy and strcat improperly

Test for NULL

- always check the return value of malloc

```
float *vector = malloc(20 * sizeof(float)); if(vector == NULL) {
            // malloc failed to allocate memory
} else {
// Process vector
}
```

Misuse of the Dereference Operator

- do not combine the dereference operator and the address of operator incorrectly

```
// bad
int num;
int *pi = &num;
// bad
int num;
int *pi;
*pi = &num;
// good
int num;
int *pi;
pi = &num;
```

Dangling Pointers

- happens when a pointer is freed but still references the memory
- reads or writes may result in runtime errors and program termination

- presents a security issue in C++ where a dangling pointer can be misused to access the virtual function table

Accessing Memory Outside the Bounds of an Array

- results in buffer overflow and should be prevented by carefully tracking the array's size

Calculating the Array Size Incorrectly

- always calculate the array's size using sizeof(array) / sizeof(type)

Misusing the sizeof Operator

- do not use the sizeof operator with a pointer that points to an array - this will only return the size of the pointer

Always Match Pointer Types

- misusing types related to pointers can cause issues (i.e. if a pointer points to a larger integer type but is cast to a smaller integer type)
- this will result in issues with cast

Bounded Pointers

- prevents pointers from accessing memory outside of the bounds of an array - must be explicitly enforced by the programmer

```
    char name[SIZE];
    char *p = name;
    if(name != NULL) {
        if(p >= name && p < name+SIZE) { // Valid pointer - continue
        } else {
        // Invalid pointer - error condition
        }
    }
```

- this can get tedious so deferring to static analysis tools, using the bounded check model (CBMC), or using C++'s smart pointers can replace this

String Security Issues

- if strcpy and strcat are not used carefully they can result in buffer overflow
- use strcpy_s and strcat_s on Microsoft (or the libsafec versions) along with scanf_s and wscanf_s
- some linux libraries support strlcpy and strlcat
- format string attacks are a type of attack where access to memory is gained by malicious user input format strings when passed to functions like printf
- look at hackerproof.org for more info
- printf, fprintf, snprintf, and syslog are all suceptible to these attacks

Pointer Arithmetic and Structures

- it is not good practice to use pointer arithmetic with the fields of a struct, even if the memory is evenly aligned

Function Pointer Issues

- be sure to use the function call operator when referencing function pointers as doing anything else may result in incorrect code functionality
- do not assign a function pointer to another when the signatures are different

Memory Deallocation Issues

Double Free

- happens when a block of memory is freed twice
- a simple solution is to always NULL a pointer
- can also use a customized free function:

```
        void saferFree(void **pp) {
            if (pp != NULL && *pp != NULL) {
```

```
            free(*pp);
            *pp = NULL;
        }
    }
    // macro
    #define safeFree(p) saferFree((void**)&(p))
```

Clearing Sensitive Data

- overwrite sensitive data in memory once the application is finished using it

  ```
      memset(name,0,sizeof(name));
      memset(securityQuestion,0,strlen(securityQuestion));
  ```

- use memset before memory deallocation

Using Static Analysis Tools

- use -Wall with GCC
- use any number of other static analysis tools

Chapter 8 - Odds and Ends

Casting Pointers

- use for:
  - Accessing a special purpose address
  - Assigning an address to represent a port
  - Determining a machine's endianness
- casting an integer to a pointer:

```
int num = 8;
int *pi = (int*)num;
```

- to access memory location 0, cast a pointer to an integer and then cast it back to a pointer:

```
pi = &num;
int tmp = (int)pi;
pi = (int*)tmp;
```

- NOTE: handles do not explicitly refer to pointers, a handle is a system resource often accessed through a pointer

Accessing a Special Purpose Address

- most hardware resources such audio and video hardware ports are accessed through special purpose addresses

```
#define VIDEO_BASE 0xB8000
int *video = (int *) VIDEO_BASE;
```

- this memory can generally be read and written to if appropriate

- to access memory location 0 (as in kernel programs) try the following: • Set the pointer to zero (this does not always work) • Assign a zero to an integer and then cast the integer to the pointer • Use a union • Use the memset function to assign a zero to the pointer

Accessing a Port

- access a port:

```
#define PORT 0xB0000000
unsigned int volatile * const port = (unsigned int *) PORT;
```

- use volatile to indicate that the data can be changed outside of the program
- volatile also prevents the runtime system from caching the port's address in a register as this makes sure each read and write is not stale
- read and write to the port:

  ```
  *port = 0x0BF4; // write to the port
  value = *port; // read from the port
  ```

- do not access volatile memory with a non-volatile variable

Accessing Memory using DMA

- DMA is system specific, outside of ANSI C, and operates using callbacks to bypass memory access through the CPU

Determining the Endianness of a Machine

```
int num = 0x12345678;
char* pc = (char*) &num;
for (int i = 0; i < 4; i++) {
    printf("%p: %02x \n", pc, (unsigned char) *pc++);
}
```

Aliasing, Strict Aliasing, and the restrict Keyword

- aliasing is when two pointers reference the same memory:

  ```
  int num = 5;
  int* p1 = &num;
  int* p2 = &num;
  ```

- aliasing prevents the ability to gain compiler optimizations by specifying that aliasing has not been used
- strict aliasing prevents a pointer of one data type from pointing to memory of a different data type
- to avoid aliasing:
    - Use a union
    - Disable strict aliasing
    - Use a pointer to char
- GCC has these options:
    - -fno-strict-aliasing to turn it off
    - -fstrict-aliasing to turn it on
    - -Wstrict-aliasing to warn of strict aliasing-related problems
- prefer resolving aliasing usage over turning off strict aliasing
- do not rely on compiler generated alias warnings

Using a Union to Represent a Value in Multiple Ways

```
typedef union _conversion {
    float fNum;
    unsigned int uiNum;
} Conversion;

int isPositive1(float number) {
    Conversion conversion = { .fNum =number};
    return (conversion.uiNum & 0x80000000) == 0;
}
```

- the above example works but replacing the data members with pointers to the same types will violate the strict aliasing rule

Strict Aliasing

- when used with structures, even if the structures types have identical fields, two pointers of the different types should not reference the same object

- however, the two pointers can reference the same object if the structure names are different, i.e.

```
    typedef struct _person { ...
    typedef Person Employee;

    Person* person;
    Employee* employee;
```

Using the restrict Keyword

- the restrict keyword signals to the compiler that the pointers are not aliased
- new code should prefer using the restrict keyword to gain better code optimizations
- it implies:
    1. To the compiler it means it can perform certain code optimizations
    2. To the programmer it means these pointers should not be aliased; otherwise, the results of the operation are undefined.
- several standard C functions use it

Threads and Pointers

- prefer POSIX threads (pthreads)

Sharing Pointers Between Threads

- use pthread_mutext_t and locking to avoid data races between threads

```
     pthread_mutex_t mutexSum;
     pthread_mutex_lock(&mutexSum);
     vectorInfo->sum += total;
     pthread_mutex_unlock(&mutexSum);
```

Using Function Pointers to Support Callbacks

```
        typedef struct _factorialData {
            int number;
            int result;
            void (*callBack)(struct _factorialData*);
        } FactorialData;

        void factorial(void *args) {
            FactorialData *factorialData = (FactorialData*) args;
            void (*callBack)(FactorialData*); // Function prototype
            int number = factorialData->number;
            callBack = factorialData->callBack;
            int num = 1;
            for(int i = 1; i<=number; i++) {
                num *= i;
            }
            factorialData->result = num;
            callBack(factorialData);
            pthread_exit(NULL);
        }

        void startThread(FactorialData *data) {
            pthread_t thread_id;
            int thread = pthread_create(&thread_id, NULL, factorial, (void *) dat
a);
        }

        void callBackFunction(FactorialData *factorialData) {
            printf("Factorial is %d\n", factorialData->result);
        }

        FactorialData *data = (FactorialData*) malloc(sizeof(FactorialData));
        if (!data) {
```

```
        printf("Failed to allocate memory\n"); return;
    }
    data->number = 5;
    data->callBack = callBackFunction;
    startThread(data);
    Sleep(2000);
```

Object-Oriented Techniques

Creating and Using an Opaque Pointer

- opaque pointers are created by declaring a type (usually a struct) within a header file but defining its implementation within the .c file

```
    // link.h
    typedef void *Data;
    typedef struct _linkedList LinkedList;

    // link.c
    typedef struct _node {
        Data* data;
        struct _node* next;
    } Node;

    struct _linkedList {
        Node* head;
    };

    // usage
    LinkedList* list = getLinkedListInstance();
    Person *person = (Person*) malloc(sizeof(Person));
    initializePerson(person, "Peter", "Underwood", "Manager", 36);
    addNode(list, person);
```

Polymorphism in C

- can add a virtual function table to a structure

```
typedef struct _shape {
    vFunctions functions; // Base variables
    int x;
    int y;
} Shape;

typedef void (*fptrSet)(void*,int);
typedef int (*fptrGet)(void*);
typedef void (*fptrDisplay)();
typedef struct _functions { // Functions
        fptrSet setX;
        fptrGet getX;
        fptrSet setY;
        fptrGet getY;
        fptrDisplay display;
} vFunctions;

typedef struct _rectangle {
    Shape base;
    int width;
    int height;
} Rectangle;

// usage with a pointer
rectangle->base.x = x;
rectangle->base.functions.display = rectangleDisplay;
rectangle->base.functions.setX = rectangleSetX;
rectangle->base.functions.getX = rectangleGetX;
rectangle->base.functions.setY = rectangleSetY;
rectangle->base.functions.getY = rectangleGetY;
```

- can continue to add more structures that derive from Shape

NOTE

- can also use the approach where public data and functions of an object are declared in .h and private data and function definition are in .c and each function takes a pointer to the structure containing the data (the this pointer) as the first argument. This mimics C++ style object orientation.