Arrays & Pointers

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss arrays and their use in C
- Explain pointers including notation, declarations, use, and dereferencing
- Discuss pointers in relation to arrays
- Explain functions use of pointers including pointers to functions
- Discuss complicated pointer declarations

Topics for today

- Arrays
  - Introduction
  - Definition
  - An Example
  - Arrays and Functions
  - More Dimensional Arrays
- Pointers
  - Pointer basics (notation and use)
  - Defining Moments, Assignments, etc
  - Pointers and Arrays
  - Pointers and Functions
  - Complicated Declarations

Arrays

- Introduction
- Definition
- An Example
- Passing Arrays to Functions
- Another Example
- More Dimensional Arrays

Array Introduction

- Arrays are ***contiguous*** group of variables with the same name and the same type
  - The contiguous is ultra-important as it is this very quality that makes arrays ultra-fast!
- The individual elements of an array are indexed such that to refer to a specific element we provide a position number of that particular element
  - Positions begin at zero in C (just like Java)
- They are referenced using subscripts
  - EX: `a[0]` refers to the 0th element
  - This subscript **must** be an integer or integer expression

Array Definition

- Arrays are defined using the following format:
  - **`type name[<number of elements>];`**
- Just as we could with other variable types we can initialize using a literal.
  - We do so using a comma separated list surrounded by braces {}
    - EX:
      - **`int x[3] = {1,2,3};`**
    - If we have too few numbers, the rest are initialized to zero (very useful!)
    - Too many numbers causes a syntax error
- Lastly, we can set the number of variables using the initializer list of literals
  - EX:
    - **`int x[] = {1,2,3};`**

An Example

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

int main() {
   int face; //die value
   int roll; //num of rolls
   int freq[SIZE] = {0};

   srand(time(NULL)); //seed random # generator

   for (roll = 1; roll <= 5000; roll++) {
      face = 1 + rand() % 6;
      freq[face]++;
   }
} //ideally it would do something with the freq array!
```


Arrays & Functions

- Arrays are passed by reference, and their elements are then passed by value
  - So if you modify the elements of the array in a function, you've modified the elements of the array
- We'll discuss when we get to pointers why this happens and mechanisms to avoid this

Gnome Sort

- Gnome Sort is based on the technique used by the standard Dutch Garden Gnome (SimpleSort)
- Here is how a garden gnome sorts a line of flower pots.
  - Basically, he looks at the flower pot next to him and the previous one;
    - if they are in the right order he steps one pot forward, otherwise he swaps them and steps one pot backwards.
  - Boundary conditions: if there is no previous pot, he steps forwards; if there is no pot next to him, he is done.

Actual Gnome Sort Code

```
void gnomesort(int n, int ar[]) {
   int i = 0;
   while (i < n) {
      if (i == 0 || ar[i-1] <= ar[i])
         i++;
      else {
         int tmp = ar[i];
         ar[i] = ar[i-1];
         ar[--i] = tmp;
      }
   }
}
```


More-Dimensions

- Sometimes (read often) you need more than one dimension
  - Images are multidimensional
    - B&W images are 2x2
    - Color images are 2x2x3
  - GIS stuff is even more dimensional
    - Longitude, latitude, depth
      - And often multiple readings such as
        - » Salinity
        - » Wave Height
        - » Etc
- So we need a way to model this with arrays

Yet Another Example

```
double avgSalinity(int area[][][], int long, int lat, int
depth) {
   int i,j,k;
   int sum = 0;
   for (i=0; i<long; i++) {
      for (j=0; j<lat; j++) {
         for (k=0; k<depth; k++) {
            sum += area[i][j][k];
         }
      }
   }
   return (sum/(long*lat*depth));
}
```

Pointers

- Pointer basics (notation and use)
- Defining Moments, Assignments, etc
- Pointers and Arrays
- Pointers and Functions
- Complicated Declarations & Other

Pointer Introduction

- Pointers are easily the most powerful feature of the C programming language
  - But the ***hardest to master***
- Pointers enable
  - Programs to simulate call by reference
  - Create and manipulate dynamic data structures
- Pointers are variables whose ***values are memory addresses***
- A variable name directly references a value whereas a pointer indirectly references a value thus referencing a value through a pointer is called indirection

Pointer Definition

- Like all variables, pointers must be defined before use
- Pointer definition follows the following syntax
  - `<pointerType> *<varname>`
  - And we say that pointer varname points to an object of pointerType

Pointer Assignment

- Pointer assignment takes one of the following forms:

/* an address (here of <someVariable> */
<pointerVarName> = &<someVariable>

/* NULL (preferred) **NULL is in stdlib*/**
<pointerVarName> = NULL;

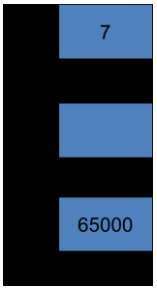
/* NULL */
<pointerVarName> = 0;

Pointer Basics (notation)

/* defining a variable */
int a = 7;

/* defining a pointer */
int *aPtr;

/*assigning to a pointer */
aPtr = &a;

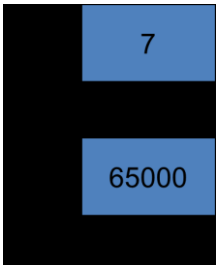


Address Operator

- & operator
  - Returns the address of a variable

/* address of aPtr */
&aPtr == 55000;

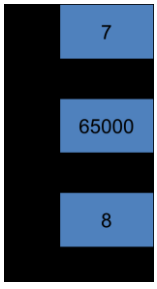
/* address of a */
&a == 65000



De-Referencing Operator

- * operator
  - defines a pointer
  - get/set the value of variable

```
/* Dereference aPtr */
*aPtr = 8;
```



Pointers and Arrays

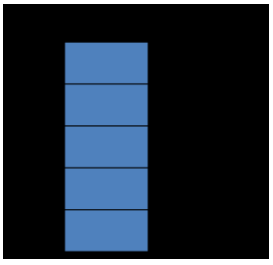
- Key points:
  - Arrays are contiguous blocks of memory in C
  - The variable of an array type can be used as the address of the 0th element
- Thus:
  - `int a[5]; int *aPtr;`
  - `aPtr = &a[0] ≈ aPtr = a;`

Pointer 'Rithmatic

- Pointer arithmetic only makes sense when using pointers as arrays
- Pointer Math Basics
  - Increment/Decrement pointers
  - Integer added to/subtracted from a pointer
  - A pointer may be subtracted from another pointer

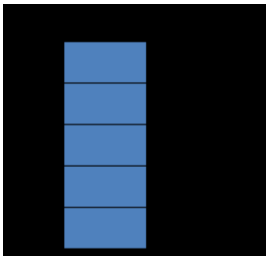
Increment/Decrement

```
int a[5]; int *aPtr; aPtr = a;
/* aPtr == 65000 */

*aPtr++ == a[1];
/* aPtr is now 65004 */

*aPtr--   == a[0];
/* aPtr is now 65000 */
```



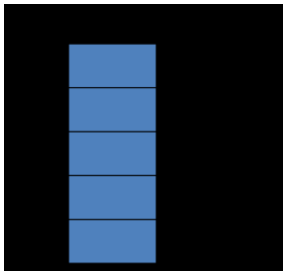
Adding an Integer to a pointer

```
int a[5]; int *aPtr; aPtr = a;
/* aPtr == 65000 */

*(aPtr + 3) == a[3];
/* aPtr is now 65012 */
```



Adding/subtracting a pointer to/from a pointer

```
int a[5]; int *aPtr;
aPtr = a; aPtr2 = &a[3];
/* aPtr == 65000 */
/* aPtr2 == 65012 */

aPtr2 – aPtr = ?
//65012 – 65000 = 12
//65012 – 65000 = 12/4 =
   3
```



Pointers & Dynamic Arrays

- Arrays by themselves are static in C
- Using pointers we can develop dynamic arrays
- For instance:
  - `int *arr;` vs `int arr[5];`
- To initialize:
  - we can pretend it's an array:
    - `int *arr = "To be or not to be";`
  - Or we can use malloc and assignment:
    - `int *arr = malloc ( sizeof(int) * 5 );`

Dynamic Memory Allocation

- **malloc**
  - `*void malloc(<sizeInBytes>)`
  - `ie:`
    - `aPtr = (int *)malloc(sizeof(int)*5)`
- **calloc**
  - `*void calloc(<number>,<baseSizeInBytes>)`
  - ie:
    - `aPtr = (int *)calloc(5,sizeof(int))`

Call-By Differences and Pointers

- Call-By-Value is default. However, this is not always ideal. Why?
- Pointers are the how for C and Call-by-reference.
- However, unrestricted access to the variable may not be ideal either. Why?
- Variations between pure call by value and pure call by reference involve the **const** keyword in C

Additional Array/Pointer Notation

- Additional but important notation
- Given: `int a[5]; int *aPtr; aPtr = a;`
- The following are all equivalent:

`const` qualifier

- Informs the compiler that the variable should not be modified
- Was not part of original C standard
- Usage will be described as we discuss the passing pointers to function mechanisms
  - Constant pointer to constant data
  - Constant pointer to non-constant data
  - Non-constant pointer to constant data
  - Non-constant pointer to non-constant data

Constant Vs Non-Constant

- Constant pointer
  - Address location pointer points to cannot be modified
  - Assuming `int *aPtr;` is passed to function
  - `aPtr = &<someOtherVariable>` is not allowed
- Constant Data
  - Data at address cannot be modified
  - Assuming `int *aPtr;` is passed to function
  - `*aPtr = <someValue>` is not allowed

Non-Constant Pointer & Data

- Highest level of data access
- No use of **`const`** qualifier, thus is the default for passing pointers to functions
- Example
  - **`void function1(int *aPtr)`**

Example of non-constant pointer & data

```
/* convert string to uppercase letters */
void convertToUppercase( char *sPtr )
{
   /* current character is not '\0' */
   while ( *sPtr != '\0' ) {
      /* if character is lowercase, */
      if ( islower( *sPtr ) ) {
         /* convert to uppercase */
         *sPtr = toupper( *sPtr ); //value change
      } /* end if */
      /* move sPtr to the next character */
      ++sPtr;  //pointer change
   } /* end while */
} /* end function convertToUppercase */
```


Example of non-constant pointer & data (cont)

```
int main()
{
   /* initialize char array */
   char string[] = "characters and $32.98";
   printf( "The string before conversion is: %s", string
);
   convertToUppercase( string );
   printf( "\nThe string after conversion is: %s\n",
string );

   return 0; /* indicates successful termination */

} /* end main */
```

Non-Constant Pointer to Constant Data

- EX:
  - `void function2(const char *sPtr)`
- Pointer can be modified, but what the pointer points to cannot.
- Process each element of an array without modifying it

Example of non-constant pointer to constant data

```
/* sPtr cannot modify the character to which it points,
   i.e., sPtr is a "read-only" pointer */
void printCharacters( const char *sPtr )
{
   /* loop through entire string */
   for ( ; *sPtr != '\0'; sPtr++ ) {   //pointer moves
      printf( "%c", *sPtr );
   } /* end for */

} /* end function printCharacters */
```


Example of non-constant pointer to constant data (cont)

```
int main()
{
   /* initialize char array */
   char string[] = "print characters of a string";

   printf( "The string is:\n" );
   printCharacters( string );
   printf( "\n" );

   return 0; /* indicates successful termination */

} /* end main */
```

Constant pointer to non-constant data

- Pointer always points to the same location in memory, and the data it points to can be modified (default for an array name)
- Must be initialized upon declaration
  - `int * const ptr = &x;`
- Typically, used in array notation use functions

Constant pointer to non-constant data EX

```
int main() {

   int x; /* define x */
   int y; /* define y */

   /* ptr is a constant pointer to an integer that can be
modified
      through ptr, but ptr always points to the same
memory location */
   int * const ptr = &x;

   *ptr = 7; /* allowed: *ptr is not const */
   ptr = &y; /* error: ptr is const; cannot assign new
address */

   return 0; /* indicates successful termination */

} /* end main */
```


Constant Pointer To Constant Data

- Least access privilege
- Always points to the same memory location and the data value at that location may not be modified
  - `const int *const ptr = &x;`
- Should be passed to a function which only accesses the pointers using array subscript notation and does not modify the array

Constant Pointer To Constant Data EX

```
int main()
{
   int x = 5; /* initialize x */
   int y;     /* define y */

   /* ptr is a constant pointer to a constant integer. ptr
       always points to the
       same location; the integer at that location cannot
       be modified */

   const int *const ptr = &x;

   printf( "%d\n", *ptr );

   *ptr = 7; // error: *ptr is const; no new value
   ptr = &y; // error: ptr is const; no new address

   return 0; /* indicates successful termination */
} /* end main */
```


Compile time errors & const

- The error message presented by attempting to modify a variable which cannot be modified is:
  - `<filename>(<lineNumber>) : error <errorNo>: l-value specifies const object`

Pointers to Functions

- A pointer can point to a function
- The pointer contains the address of the function in memory
- Very similar to array names, function names are the starting address in memory of the code that performs the function's' task
- Similar in concept to Java's Comparator
- Used in sorting, menu driven applications, etc

Pointers to Function EX (1)

```
/* prototypes */
void bubble( int work[], const int size,
             int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );
void swap( int *element1Ptr, int *element2ptr );

int ascending( int a, int b )
{
   return b < a;   /* swap if b is less than a */

} /* end function ascending */

int descending( int a, int b )
{
   return b > a;   /* swap if b is greater than a */

} /* end function descending */
```


Pointers to Function EX (2)

```
void bubble( int work[], const int size,
             int (*compare)( int a, int b ) )
{
   int pass;  /* pass counter */
   int count; /* comparison counter */

   for ( pass = 1; pass < size; pass++ ) { //pass loop
      //comparison loop
      for ( count = 0; count < size - 1; count++ ) {
         //if out of order, swap
         if ( (*compare)( work[count], work[count + 1])) {
            swap( &work[ count ], &work[ count + 1 ] );
         }
      }
   }
}
```


Pointers to Function EX (3)

```
int main()
{
   int order, int counter;

   order = 1;

   /* initialize array a */
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
   }
   else { /* pass function descending */
      bubble( a, SIZE, descending );
   }
}
```

Complicated Declarations

- The Key to decipher complicated declarations are C's operator precedence rules
- Does everyone have a copy of these?
  - If not, I'll post them.
- Some examples of complicated declarations:
  - `char (*(*x())[])()`
    - x: function returning pointer to array[] of pointer to function returning char
  - `char(*(*x[3])())[5]`
    - x: array[3] of pointer to function returning pointer to array[5] of char
- I am sorry to inform you that these exist in reality

Summary

- Discussed arrays and their use in C
- Explained pointers including notation, declarations, use, and dereferencing
- Discussed pointers in relation to arrays
- Explained functions use of pointers including pointers to functions
- Discussed complicated pointer declarations

Questions?