Agenda

1 C Pointers

- 1.1 Introduction
- 1.2 Pointer Operators
- 1.3 Passing Arguments to Functions by Reference
- 1.4 Using the const Qualifier with Pointers
- 1.5 Pointer Expressions and Pointer Arithmetic
- 1.6 Relationship Between Pointers and Arrays
- 1.7 Arrays of Pointers
- 1.8 Pointers to Functions

C Pointers
Introduction

- Pointers are among C's most difficult capabilities to master. Pointers enable program to simulate pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures that can grow and shrink at execution time.

C Pointers

Pointer Variable Definitions and Initialization

- Pointers are variables whose values are memory addresses. Normally, a variable directly contains a specific value. A pointer, on the other hand, contains an address of a variable that contains a specific value.
- In this sense, a variable name directly references a value, and a pointer indirectly references a value. Referencing a value through a pointer is called indirection.
- Pointers, like all variables, must be defined before they can be used. The definition

  int *countPtr;

  specifies that variable countPtr is of type int * (a pointer to an integer) and is read right to left, countPtr is a pointer to in or countPtr points to an object of type int.

Good Programming Practice

We prefer to include the letters Ptr in pointer variable names to make it clear that these variables are pointers and thus need to be handled appropriately.

Pointer Variable Definitions and Initialization

- Pointers should be initialized when they're defined, or they can be assigned a value. A pointer may be initialied to NULL, 0 or an address.

- A pointer with the value NULL points to nothing. NULL is a symbolic constant defined in the <stddef.h> header. Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred.

Error-Prevention Tip
Initialize pointers to prevent unexpected results.

Pointer Operators

- The, &, or address operator, is a unary operator that returns the address of its operand. For example, assuming the definitions

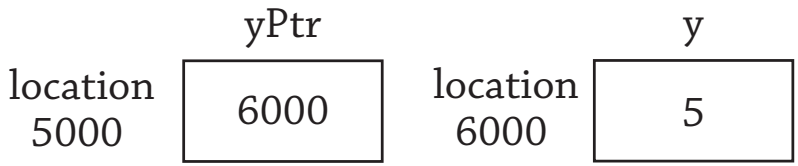
```
int y = 5;
int *yPtr;
```

and statement

```
yPtr = &y;
```

assigns the address of the variable y to pointer variable yPtr. Variable yPtr is then said to point to y.

- Figure below show the representation of the pointer in memory, assuming the integer variable y is stored and location 6000, and pointer varaible yPtr is stored at location 5000. The operand of the address operator must be a variable, the address operator cannot be applied to constants or expressions.



Pointer Operators

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the value of the object to which its operand points. For example, the statemet

  printf(“%d”, *yPtr);

  prints the value of variable y, namely 5. Using * in this manner is called dereferencing a pointer.

Common Programming Error

Dereferencing a pointer that has not been properly initialized or that has not been assigned to point to a specific location in memory is an error. This could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion with incorrect results.

Pointer Operators

- Figure below demonstrates the pointer operators & and *. The printf conversion specifier %p outputs the memory location as a hexadecimal integer on most platforms.

- The & and * operators are complements of one another, whey they're both applied consequtively to aPtr in either order the same result is printed.

```
1   // Using the & and * pointer operators
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      int a = 5; // a is an integer
8      int *aPtr = &a; // aPtr is a pointer to an integer
9
10     printf( "The address of a is %p\n", &a );
11     printf( "The value of aPtr is %p\n", aPtr );
12
13     printf "The value of a is %d\n", a );
14     printf( "The value of *aPtr is:%d\n", *aPtr );
15
16     printf( "&*aPtr:%p\n", &*aPtr );
17     printf( "*&aPtr:%p\n", *&aPtr );
18
19     return 0;
20  } // end main
```

- Figure below lists the precedence and associativity of the operators introduced to this point.

[] () ++(postfix) --(postfix)	left to right	highest
+ - ! ++(prefix) --(prefix) (type) & *	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

C Pointers

Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function, pass-by-value and pass-by-reference. All arguments in C are passed by value.

- Many functions require the capability to modify variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value.

- In C, you use pointers and the indirection operator to simulate pass-by-reference. When calling a function with arguments that should be modified, the addresses of the arguments are passed.

Passing Arguments to Functions by Reference-Pass-By-Value

- The program below passes the variable number by value to function cubeByValue. The cubeByValue function cubes its argument and passes the new value back to main using a return statement.

```
1   // Cube a variable using pass-by-value
2   #include <stdio.h>
3
4   int cubeByValue( int n ); //prototype
5
6   // function main begins program execution
7   int main( void )
8   {
9     int number = 5;
10    printf( "The original value of number is %d\n", number );
11
12    number = cubeByValue( number );
13    printf( "The new value of number is %d\n", number );
14
15    return 0;
16  } // end main
17
18  int cubeByValue( int n )
19  {
20    return n * n * n;
21  } // end main
```

Passing Arguments to Functions by Reference-Pass-By-Reference

The program below passes the variable number by reference to function cubeByFunction. The cubeByReference function takes as a parameter a pointer to an int called nPtr. The function dereferences the pointer and cubes the value to which nPtr points, then assigns the result to *nPtr, thus changing the value of number in main.

```
1   // Cube a variable using pass-by-reference
2   #include <stdio.h>
3
4   int cubeByReference( int *nPtr ); //prototype
5
6   // function main begins program execution
7   int main( void )
8   {
9     int number = 5;
10    printf( “The original value of number is %d\n”, number );
11
12    cubeByReference( &number );
13    printf( “The new value of number is %d\n”, number );
14
15    return 0;
16  } // end main
17
18  int cubeByValue( int n )
19  {
20    *nPtr = *nPtr * *nPtr * *nPtr;
21  } // end main
```

Passing Arguments to Functions by Reference-Pass-By-Reference

- The function prototype for cubeByReference contains int * in parantheses.
- For a function that expect a single-subscripted array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function cubeByReference.
- The compiler does not differentiate between a function that receives a pointer and one that receives a single-subscripted array. When the compiler encounters a function parameter for a single-subscripted array of the form int b[], the compiler converts the parameter to the pointer notation int *b. The two forms are interchangeable.

Using the const Qualifier with Pointers

- The const qualifier enables you to inform the compiler that the value of a particular variable should not be modified.

- Six possibilities exist for using const with funciton parameters, two with pass-by-value parameter passing and four with pass-by-reference parameter passing. How do you choose one of the six possibilities ? Let the principle of least privilege be your guide. Always award a function enough access to the data in its parameters to accomplish its specified task, but absolutely no more.

- There are four ways to pass a pointer to a function: a non-constant pointer to non-constant data, a constant pointer to non-constant data, a non-constant pointer to constant data, and a constant pointer to constant data. Each of the four combinations probides different access privileges.

Converting a String to Uppercase Using a Non-Constant Pointer to Non-Constant Data

- The highest level of data access is granted by a non-constant pointer to non-constant data. In this case, the data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.

```
1   // non-constant pointer to non-constant data
2   #include <stdio.h>
3   #include <ctype.h>
4
5   void convertToUppercase( char *sPtr ); // prototype
6
7   // function main begins program execution
8   int main( void )
9   {
10    char string [] = "cHaRaCters and $32.98";
11    printf( "The string before conversion is: %s\n", string );
12
13    convertToUppercase( string );
14    printf( "The string after conversion is :%s\n", string );
15
16    return 0;
17  } // end main
18
19  void convertToUppercase( char *sPtr ){
20  {
21    while( *sPtr != '\0' ){
22      *sPtr = toupper( *sPtr );
23      sPtr = sPtr + 1;
24    }
25  } // end main
```

Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data

- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points cannot be modified. Such a pointer might be used to receive an array argument to a function that will process each element without modifying the data.

```
1   // non-constant pointer to constant data
2   #include <stdio.h>
3
4
5   void printCharacters( const char *sPtr ); // prototype
6
7   // function main begins program execution
8   int main( void )
9   {
10    char string [] = “print characters of a string”;
11    puts( “The string is:” );
12    printCharacters( string )
13    puts( “” );
14
15    return 0;
16  } // end main
17
18  void printCharacters( char *sPtr ){
19  {
20    for( ; *sPtr != ‘\0’; ++sPtr ){
21      printf( “%c”, *sPtr );
22    }
23  } // end main
```

Printing a String One Character at a Time Using a Non-Constant Pointer to Constant Data

- For example, function printCharacters declares paraeter sPtr to be of type const char *. The declaration is read from right to left as “sPtr is a pointer to a character constant”.

- The function uses a for statement to output each character in the string until the null character is encountered. After each character is printed, pointer sPtr is incremented to point to the next character in the string.

- The figure below illustrates the attempt to compile a function that receives a non-constant pointer to constant data. This function attempts to modify the data pointed to by xPtr in line 16, which results in a compilation error.

```
1   // non-constant pointer to constant data
2   #include <stdio.h>
3
4   void f( const int *xPtr ); // prototype
5
6   // function main begins program execution
7   int main( void )
8   {
9     int y;
10    f( &y );
11    return 0;
12  } // end main
13
14  void printCharacters( char *sPtr ){
15  {
16    *xPtr = 100; // error: cannot modify a const object
17  } // end main
```

Attempting to Modify a Constant Pointer to Non-Constant Data

- A constant pointer to non-constant data always points to the same memory location, and the data at that location can be modified through the pointer. This is the default for an array name.

- Pointers that are declared const must be initialized when they're defined. Pointer ptr is defined in line 9 to be of type int * const. The definition is read from right to left as "ptr is a constant pointer to an integer".

```
1   // constant pointer to non-constant data
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      int x;
8      int y;
9      int * const ptr = &x
10     *ptr = 7;
11     *ptr = &y; // error: ptr is const, cannot assign new address
12     return 0;
13  } // end main
```

Attempting to Modify a Constant Pointer to Constant Data

- The least access privilege is granted by a constant pointer to constant data. Such a pointer always points to the same memory location, and the data at that memory location cannot be modified.

- This is how an array should be passed to a function that only looks at the array using array subscript notation and does not modify the array. The figure below defines a pointer variable ptr to be of type const int *const, which is read from right to left as "ptr is a constant pointer to an integer constant".

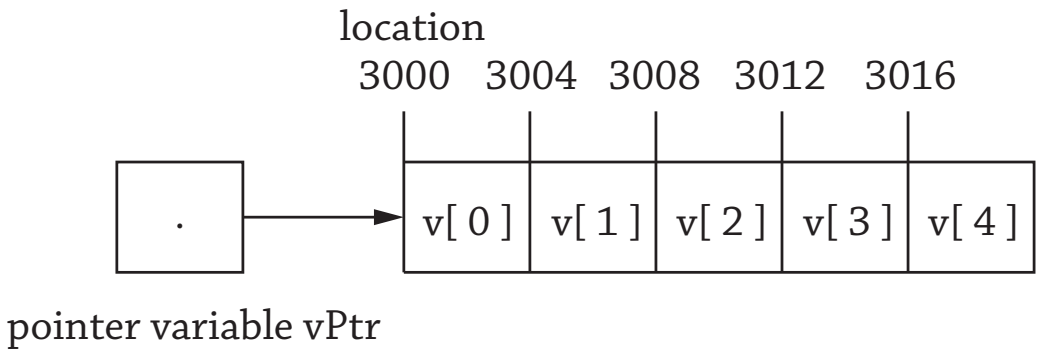
```
1   // constant pointer to constant data
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7     const int * const ptr = &x
8     printf( "%d\n", *ptr );
9     *ptr = 7; // error: ptr is const, cannot assign new value
10    *ptr = &y; // error: ptr is const, cannot assign new address
11
12    return 0;
13  } // end main
```

Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expression and comparison expressions. However, not all the operators normally used in these expressions are valid in conjunction with pointer variable.

- A limited set of arithmetic operations may be performed on pointers. A pointer may be incremented (++) or decremented (--), an integer may be added to a pointer (+ or +=), an integer may be subtracted from a pointer (- or -=) and one pointer may be subtracted from another, this last operations is meaningful only when both pointers point to elements of the same array.

- Assume that array int v [5] has been defined and its first element is at location 3000 in memory. Assume vPtr has been initialized to point to v[0], the value of vPtr is 3000. The figure below illustrates this situation for a machine with 4-byte integers. Variable vPtr can be initialized to point to array v with either of statements vPtr = v; or vPtr = &v[0].



Pointer Expressions and Pointer Arithmetic

- In conventional arithmetic, 3000 + 2 yields the value 3002. This is normally not he case with pointer arithmetic. When an integer is added to or subtracted from a pointer, the pointer is not incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.

- For example, the statement vPtr += 2 would produce 3008 (3000 + 2 * 4), assuming an integer is stored in 4 bytes of memory. In the array v, vPtr would now point to v[2]. If an integer is stored in 2 bytes of memory, then the preceding calculation would result in memory location 3004 (3000 + 2 * 2).

- If a pointer is being incremented or decremented by one, the increment (++) and decrement (--) operators can be used. Either of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next location in the array. Either of the statements

```
--vPtr;
vPtr--;
```

decrements the pointer to point to the previous element of the array.

Pointer Expressions and Pointer Arithmetic

- A pointer can be assigned to another pointer if both have the same type. The exception to this rule is the pointer to void (void*), which is a generic pointer that can represent any pointer type. All pointer types can be assigned a pointer to void, and a pointer to void can be assigned a pointer of any type. In both cases, a cast operation is not required.

- A pointer to void cannot be dereferenced. Consider this: the compiler knows that a pointer to int refers to 4 bytes of memory on a machine with 4-byte integers, but a pointer to void simply contains a memory location for an unknown data type, the precise number of bytes to which the pointer refers is not known by the compiler.

- Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the same array. Pointer comparisons compare the address stored in the pointers. A common use of pointer comparison is determining whether a pointer is NULL.

C Pointers

Relationship Between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably. An array name can be thought of as a constant pointer. Pointers can be used to do any operation involving array subscripting.

- Assume that integer array b[5] and integer pointer variable bPtr have been defined. Because array name is a pointer to the first element of the array, we can set bPtr equal to the address of the first element in array b with the statement bPtr = b.

- Array element b[3] can alternatively be referenced with the pointer expression *(bPtr + 3). The 3 in the expression is the offset to the pointer. When the pointer points to the array's first element, the offset indicates which array element should be referenced, and the ofset value is identical to the array subscript. This notation is referred to as pointer/offset notation.

- The array itself can be treated as a pointer and used in pointer arithmetic. For example, the expression *(b + 3) also refers to the array element b[3].

- Pointers can be subscripted like arrays. If bPtr has the value b, the expression bPtr[1] refers to the array element b[1]. This is referred to as pointer/subscript notation.

Relationship Between Pointers and Arrays

- Figure below uses four methods we've discussed for referring to array elements, array subscripting, pointer/offset with array name as a pointer, pointer subscripting and pointer/offset with a pointer.

```
1   // Using subscripting and pointer notations with arrays.
2   #include <stdio.h>
3   #define ARRAY_SIZE 4
4
5   // function main begins program execution
6   int main( void )
7   {
8      int b [] = { 10, 20, 30, 40 };
9      int *bPtr = b;
10     size_t i;
11     size_t offset;
12
13     for( i = 0; i < ARRAY_SIZE; i++ ){
14        printf( "b[ %u ] = %d\n", i, b[ i ] );
15     {
16
```

```
17      for( offset = 0; offset < ARRAY_SIZE; offset++ ){
18        printf( "*( b + %u ) = %d\n", offset, *( b + offset )  );
19      {
20
21      for( i = 0; i < ARRAY_SIZE; i ++ ){
22        printf(  "bPtr[ %u ] = %d\n", i, bPtr[ i ] );
23      {
24
25      for( offset = 0; offset < ARRAY_SIZE; offset++ ){
26        printf( "*( bPtr + %u ) = %d\n", offset, *( bPtr + offset )  );
27      {
28      return 0;
29    } // end main
```

Arrays of Pointers

- Arrays may contain pointers. A common use of an array of pointers is to form an array of strings, referred to simply as a string array. Each entry in the array is a string, but in C a string is essentially a pointer to its first character.

- Consider the definition of string array suit, which might be useful in representing a deck of cards.

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

- The four values to be placed in the array are “Hearts”, “Diamonds”, “Clubs” and “Spades”. Each is stored in memory as a null-terminated character string that’s one character longer than the number of characters between quotes. The four strings are 7, 9, 6 and 7 characters long, respectively.

- Each pointer points to the first character of its corresponding string. Thus, even though the suit array is fixed in size, it provides access to character strings of any length. This flexibility is one example of C’s powerful data-structuring capabilities.

Pointers to Function

- A pointer to function contains the address of the function in memory. We saw that an array name is really the address in memory of the first element of the array. Similary, a function name is really the starting address in memory of the code that performs the function's task.

- Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

```
1   // Pointers to function.
2   #include <stdio.h>
3   void func( int paramter );
4
5   // function main begins program execution
6   int main( void )
7   {
8      void ( *funcPtr )( int ) = &func
9      ( *funcPtr )( 10 );
10     return 0;
11  } // end main
12
13  void func( int parameter ){
14     printf( "Value:%d\n", parameter );
15  }
```

```
1   // Pointers to function.
2   #include <stdio.h>
3   #include <stdlib.h>
4
5   #define ARRAY_SIZE 5
6   int findMin( int *array );
7   void printMin( int ( *func )( int* ), int *array );
8   // function main begins program execution
9   int main( void )
10  {
11     int array[ ] = { 40, 30, 10, 20 ,60 };
12     printMin( &findMin, array );
13     return 0;
14  } // end main
15
16  int findMin( int *array ){
17     int minValue = *array;
18     for( int i = 1; i < ARRAY_SIZE; i++ ){
19        if( minValue > *( array + i ) ){
20           minValue = *( array + i );
21        }
22    }
23    return minValue;
24  }
25  void printMin( int ( *func )( int* ), int *array  ){
26     printf( "Min:%d\n", ( *func )(  array ) );
27  }
```