COP 3014 – Programming I

Chapter 7 – Pointers

What is a Pointer

- Basic definition
  - Normally a variable contains a specific value
  - A pointer on the other hand contains the memory address of a variable which, in turn, contains a specific value
  - A pointer is ***a variable that stores an address***
  - Used to store the addresses of other variables
- A variable name directly references a value
- A pointer indirectly references a value
- Referencing a value through a pointer is referred to as **indirection**

Principle of Least Privilege – code should be granted only the amount of privilege and access needed to accomplish its task, but no more.

Declaring Pointers

- Pointer declarations use * following the type to declare.
- In a declaration, * isn't an operator, it is there to indicate that variable being declared is a pointer
- Declared by placing an asterisk (*) before the variable name

```
typeName* variableName;
```

- The following is read from right to left as, *countPtr is a pointer to int*

```
int count; //declaration of variable count
int* countPtr; //declaration of pointer countPtr
```

Declaring Pointers

- The notation can be confusing because of the placement of *.
- The following three declarations are identical. They all declare `countPtr` as a pointer to an int.

```
int* countPtr;
int *countPtr;
int * countPtr;
```

- We can declare multiple variables of the same type on one line, but for a pointer you must include the * operator for each.

```
int x, y, z;   //declaration of three variables of type int
int* p, q, r;  //appears to declare three pointers to ints, but actually
               //creates one pointer and two ints.
int * p, * q, * r; //correct way to declare three pointers to ints on one line
```

Pointers – Initializing

- Pointers should be initialized to 0, NULL or an address when declared or in an assignment
- NULL Pointer

```
int* yPtr;
yPtr = 0;
--OR-
int* yPtr = 0;
```

  - 0 is the only integer literal that can be assigned to a pointer
  - A pointer with the value 0 or NULL points to nothing, i.e. **null pointer**
  - Initializing a pointer to NULL is equivalent to initializing it to 0
  - In C++, 0 is used by convention
  - Typically a placeholder to initialize pointers until their actual values are known

Pointers – Initializing

- NULL Pointer continued…
  - Initializing pointers prevents accessing unknown or uninitialized areas of memory
  - If a pointer's value is unknown, it will likely be random memory garbage and unsafe to dereference.
  - Don't try to dereference a null pointer – results in a segmentation fault
  - If you always set pointers to null or another valid target, you can test prior to dereferencing as in,

```
if (yPtr != 0) // safe to dereference
  std::cout << *yPtr;
```

Pointer Operators

- address operator `&`
- indirection or dereferencing operator `*`

Pointer Operators – Address

- Address-of operator **`&`**
  - Unary operator
  - Obtains the memory address of its operand
  - Below, **`&y`** means *"address of y"*

```
int y = 5; // declare variable y
int* yPtr; // declare pointer variable yPtr
yPtr = &y; // assign address of y to yPtr
```

  - **Note:** Not the same as **`&`** in a reference variable declaration which is always preceded by a data-type name.
    When declaring a reference, the **`&`** is part of the type.

```
int& count;
```

Pointer Operators – Dereference

- indirection or dereferencing operator *
  - Returns an alias for the object to which its pointer operand points
  - In the declaration statement, the type appears before the *

```
std::cout << *yPtr << std::endl; // prints the value of y
--just as,
std::cout << y << std::endl; // prints the vlaue of y
```

  - After the declaration statement, the * is **dereferencing the pointer** to obtain the value
  - You can return the actual data item or value by ***dereferencing the pointer***

```
std::cout << "The data value is " << *yPtr; // prints the value
```

Pointer Operators – Dereference

- Repeating the code snippet from above, for simplicity suppose the address stored is 1234 (also see Fig 7.4 in text)

```
int y = 5; // declare variable y
int* yPtr; // declare pointer variable yPtr

yPtr = &y; // assign address of y to yPtr

std::cout << "The pointer is: " << yPtr; // prints the pointer
std::cout << "The data value is " << *yPtr; // prints the value

// Output
// The pointer is: 1234   // actual output depends on address
// The value is: 5
```

- Can also be used on the left side of an assignment statement. The following assigns 9 to y:

```
*yPtr = 9;
```

Pointers – Sample Executable

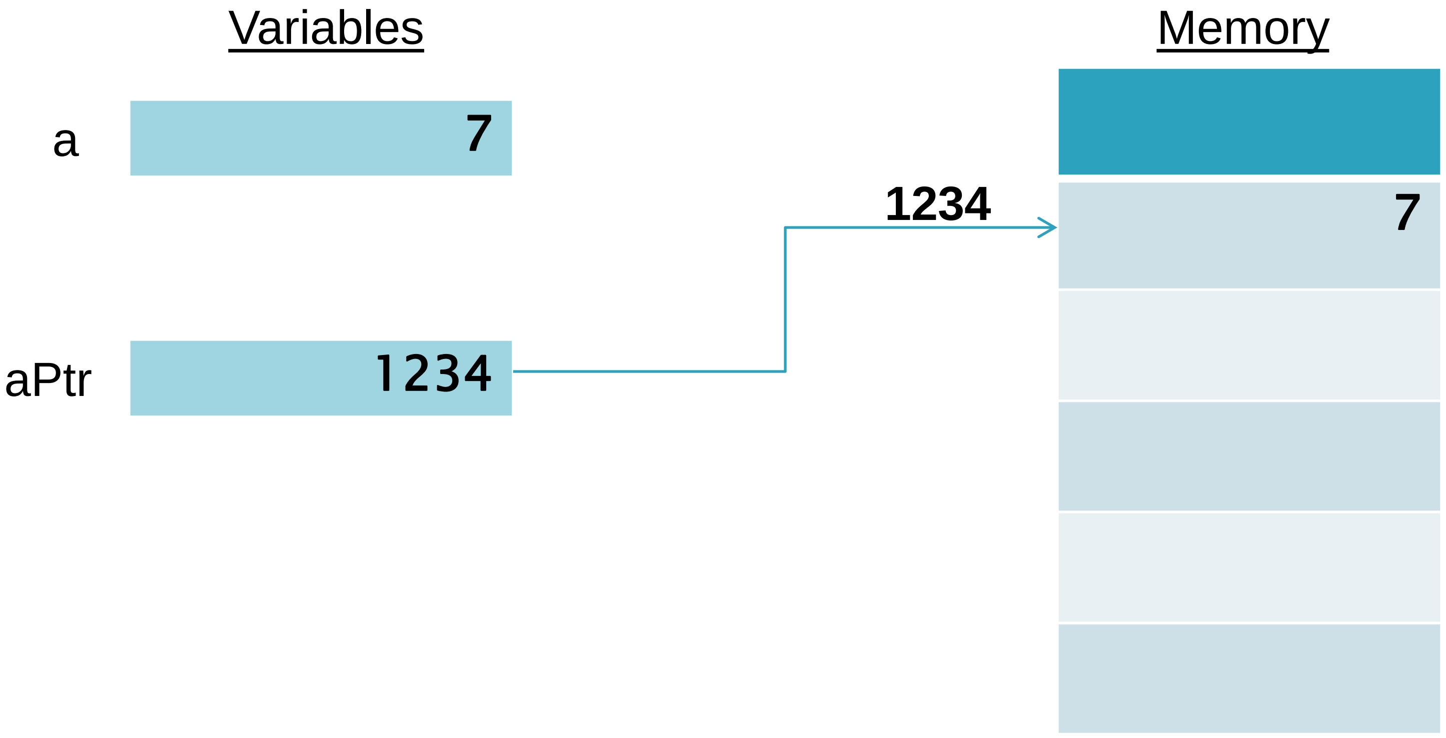
```
// Fig. 7.4: fig07_04.cpp
// Pointer operators & and *.
#include <iostream>

{
  int a; // a is an integer
  int* aPtr; // aPtr is an int * which is a pointer to an integer

  a = 7; // assigned 7 to a
  aPtr = &a; // assign the address of a to aPtr

  std::cout << "The address of a is " << &a
            << "\nThe value of aPtr is " << aPtr;
  std::cout << "\n\nThe value of a is " << a
            << "\nThe value of *aPtr is " << *aPtr;
  std::cout << "\n\nShowing that * and & are inverses of "
            << "each other.\n&*aPtr = " << &*aPtr
            << "\n*&aPtr = " << *&aPtr << std::endl;
} // end main
```

Pointers – Memory



3 ways to pass arguments to functions

- Pass-by-value
- Pass-by-reference with reference parameters
- Pass-by-reference with pointer parameters

- In an earlier lecture I used an actual reference parameter which was easier to see that it was passed as a reference in the call. Consider the following

```
void cubeByReference( int & ); // prototype
…
int number = 5;

cubeByReference( number ); // pass number by reference void

cubeByReference( int& number )
```

Pointers: Pass-by-Reference w/Pointers

- Use pointers and the dereference operator * to pass-by-reference
- When calling a function with an argument that should be modified, pass the address
- The style of the call clearly indicates pass-by-reference, as opposed to a non-pointer reference
- The name of an array is the address of the first element of that array
- ***Direct access to value – modifies value directly***
- A function receiving an address as an argument must define a pointer to receive the address. (see the function header in following example)
- Following two slides compares pass-by-value to pass-by-reference

Pointers: Pass-by-Value

```
// Fig. 7.6: fig07_06.cpp
// Pass-by-value used to cube a variable
#include <iostream>

int cubeByValue( int ); // prototype

int main()
{
  int number = 5;

  std::cout << "The original value of number is " << number;

  number = cubeByValue( number ); // pass number by value to cubeByValue
  std::cout << "\nThe new value of number is " << number << std::endl;
} // end main

// calculate and return cube of integer argument
int cubeByValue( int n )
{
  return n * n * n; // cube local variable n and return result
} // end function cubeByValue
```

Pointers: Pass-by-Reference

```
// Fig. 7.7: fig07_07.cpp
// Pass-by-reference with a pointer argument used to cube a variable's value
#include <iostream>

void cubeByReference( int* ); // prototype

int main()
{
  int number = 5;

  int* ptrNumber = &number;
  std::cout << "The address of number &number is " << &number << '\n';
  std::cout << "The address stored in ptrNumber is " << ptrNumber << "\n\n";
  std::cout << "The original value of number is " << number;

  cubeByReference( &number ); // pass number address to cubeByReference
  // cubeByReference( ptrNumber) is identical, a pointer is an address

  std::cout << "\nThe new value of number is " << number << std::endl;
} // end main

// calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int* nPtr )
{
  *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
} // end function cubeByReference
```

Pointers: constness

- *Principle of Least Privilege*
- The use of const enables you to inform the compiler that the value of a particular variable should **NOT** be modified
- Four ways to pass a pointer to a function:
  - Nonconstant Pointer to Nonconstant Data
  - Nonconstant Pointer to Constant Data
  - Constant Pointer to Nonconstant Data
  - Constant Pointer to Constant Data

Pointers: constness

- Nonconstant Pointer to Nonconstant Data
  - Highest access granted
  - Data can be modified through the dereferenced pointer
  - Pointer can be modified to point to other data
  - Read from right to left as *"countPtr is a pointer to an integer“*

```
int* countPtr;
```

Pointers: constness

- Nonconstant Pointer to Constant Data
  - Pointer can be modified to point to other data
  - The data to which it points can **NOT** be modified
  - Useful when passing an array to a function that will access all elements of the array but shouldn't modify the data
  - Read from right to left as *"countPtr is a pointer to a constant integer"*

```
const int* countPtr;
```

Fig. 7.10

Pointers: constness

```
// Fig. 7.10: fig07_10.cpp
// Attempting to modify data through a
// nonconstant pointer to constant data.

void f( const int * ); // prototype

int main()
{
   int y;

   f( &y ); // f attempts illegal modification
} // end main

// xPtr cannot modify the value of constant variable to which it points
void f( const int *xPtr )
{
   *xPtr = 100; // error: cannot modify a const object
} // end function f
```

Pointers: constness

- Constant Pointer to Nonconstant Data
  - Pointer always points to the same memory location
  - Data can be modified
  - Pointer can **NOT** be modified to point to other data
  - Since the pointer is const it must be initialized when declared
  - An example is array name which is a constant pointer to the beginning of the array
  - Read from right to left as *"countPtr is a constant pointer to a nonconstant integer"*

```
int* const countPtr = &x; //const pointer must be initialized when declared
```

Fig. 7.11

Pointers: constness

```
// Fig. 7.11: fig07_11.cpp
// Attempting to modify a constant pointer to nonconstant data.

int main()
{
   int x, 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; // const pointer must be initialized

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign to it a new address
} // end main
```

Pointers: constness

- Constant Pointer to Constant Data
  - Minimum access granted
  - Pointer always points to the same memory location
  - Data can **NOT** be modified
  - Pointer can **NOT** be modified to point to other data
  - Since the pointer is const it must be initialized when declared
  - Read from right to left as *"countPtr is a constant pointer to a constant integer"*

```
const int* const countPtr = &x; //const pointer must be initialized when declared
```

Fig. 7.12

Pointers: constness

```
// Fig. 7.12: fig07_12.cpp
// Attempting to modify a constant pointer to constant data.
#include <iostream>

int main()
{
   int x = 5, 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;

   std::cout << *ptr << std::endl;

   *ptr = 7; // error: *ptr is const; cannot assign new value
   ptr = &y; // error: ptr is const; cannot assign new address
} // end main
```

Pointer Arithmetic

- **sizeof** Operator
  - determines the size of any data type, variable or constant in bytes during program compilation
  - When applied to the name of an array, it returns the total number of bytes in the array
    - Return value is of type **`size_t`** which is an unsigned integer at least as big as `unsigned int`

Fig 7.14 and 7.15

Pointer Arithmetic

- Certain arithmetic operations may be performed on pointers
- Pointer arithmetic is only meaningful when performed on pointers that point to an array
- Arithmetic Operations
  - Incremented (++) or Decremented (--)
  - Integer may be added to (+ or +=) or subtracted from (- or -=)
  - Within contiguous data sets such as an array, one pointer may be subtracted from another of the same type resulting in the number of elements between the two
- Operations are not literal but instead add or subtract the number of units

Relationship between Pointers and Arrays

- An array name can be thought of as a constant pointer
- Array name (without subscript) points to first element of array
- Pointers can be used to do any operation involving array subscripting
  - Assume the following declarations:

```
int b[5];  // create a 5-element int array b
int* bPtr; // create int pointer bPtr
```

  - Assigning addresses

```
bPtr = b;       // assigns address of array b to bPtr
bPtr = &b[0];   // also assigns address of array b to bPtr
```

  - Pointer/Offset notation

```
*( bPtr + 3 )   // alternate way to access array element b[3]
*( b + 3 )      // also refers to element 3; using pointer arithmetic
```

Fig. 7.18

C-style Strings

- Hidden Assumptions
  - Null-terminated
  - Memory has been allocated
- Character arrays
- Arrays of type `char` terminated with the null character ‘`\0`’
- Null character ‘`\0`’ marks where the string terminates in memory.
- Must allocate one extra space for the null terminator ‘`\0`’ in the last element in arrays of characters that are used as strings
- More memory management required due to null character
- C string functions do not handle or "worry about" memory management
- As a type, a C-String is nothing more than a pointer to `char`
- Common in older legacy code

C-style Strings

- String literal
  - Enclosed in double quotes
  - Compiler allocates enough memory for a string , including the null terminator.

```
char name[ ] = "Tony";  //"Tony" is a string literal
```

  - The above array, `name` has a size of five characters.
  - An empty string (“”) actually has space reserved for the null terminator.
  - Both of the following create a five-element array color containing the characters ‘b’, ‘l’, ‘u’, ‘e’, and ‘\0’

```
char color[] = "blue";
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

  - ***ADVANCED*** but for comparison:

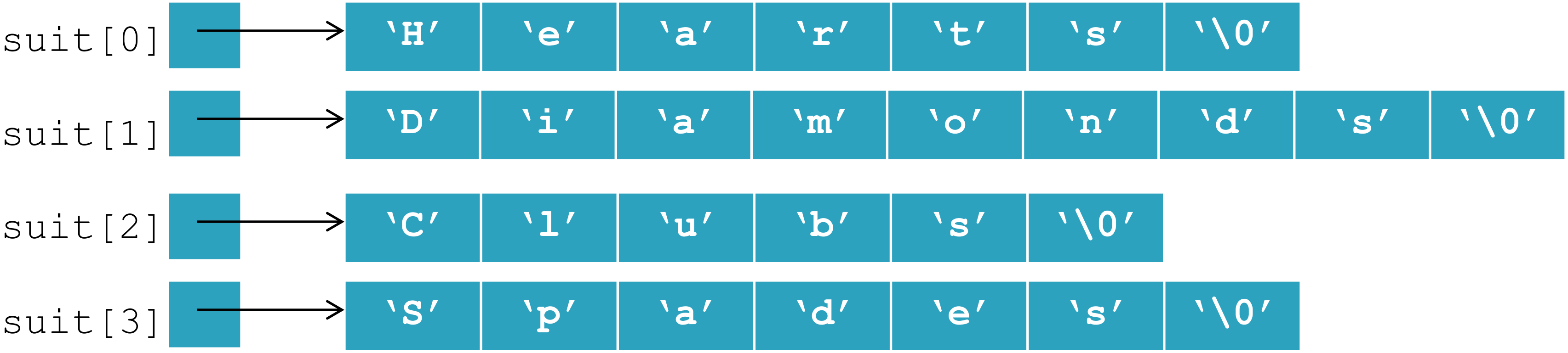
```
const char* colorPtr = "blue";
```

    - Creates a pointer variable `colorPtr` that points to the letter ‘b’ in the string “blue” (which ends in ‘\0’) and resides somewhere in memory

Array of Pointers

- Arrays may contain pointers
- String Array – array of pointer-based strings

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



suit[0]	‘H’	‘e’	‘a’	‘r’	‘t’	‘s’	‘\0’		
suit[1]	‘D’	‘i’	‘a’	‘m’	‘o’	‘n’	‘d’	‘s’	‘\0’
suit[2]	‘C’	‘l’	‘u’	‘b’	‘s’	‘\0’			
suit[3]	‘S’	‘p’	‘a’	‘d’	‘e’	‘s’	‘\0’		

- Pointer to constant char data
- Stored as null-terminated character strings, one character longer than the literal
- Only pointers are stored in the array, not strings

Questions?