Lecture 17:
Pointers

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 26th, 2010

8.1 Introduction

- Pointers also enable pass-by-reference and can be used to create and manipulate dynamic data structures that can grow and shrink, such as linked lists, queues, stacks and trees.
- This chapter explains basic pointer concepts and rein-forces the intimate relationship among arrays and pointers.

8.2 Pointer Variable Declarations and Initialization

- A pointer contains the memory address of a variable that, in turn, 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.

8.2 Pointer Variable Declarations and Initialization (cont.)

- The declaration
  - `int *countPtr, count;`

  declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value) and is read (right to left), "`countPtr` is a pointer to `int`."
  - Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
  - The * in the declaration applies only to `countPtr`.
  - Each variable being declared as a pointer must be preceded by an asterisk (*).
- When * appears in a declaration, it isn't an operator; rather, it indicates that the variable being declared is a pointer.
- Pointers can be declared to point to objects of any data type.

8.2 Pointer Variable Declarations and Initialization (cont.)

- Pointers should be initialized either when they're declared or in an assignment.
- A pointer may be initialized to `0`, `NULL` or an address of the corresponding type.
- A pointer with the value `0` or `NULL` points to nothing and is known as a null pointer.
  - `NULL` is equivalent to `0`, but in C++, `0` is used by convention.
- The value `0` is the only in-teger value that can be assigned directly to a pointer variable without first casting the integer to a pointer type.

8.3 Pointer Operators

- The address operator (&) is a unary operator that obtains the memory address of its operand.
- Assuming the declarations
  - ```
    int y = 5; // declare variable y
    int *yPtr; // declare pointer variable
    yPtr
    ```

  the statement
  - ```
    yPtr = &y; // assign address of y to yPtr
    ```

  assigns the address of the variable `y` to pointer variable `yPtr`.
- Figure 8.2 shows a schematic representation of memory after the preced-ing assignment.

8.3 Pointer Operators (cont.)

- Figure 8.3 shows another pointer representation in memory with integer variable `y` stored at memory location `600000` and pointer variable `yPtr` stored at memory location `500000`.
- The operand of the address operator must be an *lvalue*; the address operator cannot be applied to constants or to expressions that do not result in references.
- The * operator, commonly referred to as the indirection operator or dereferencing operator, returns a synonym for the object to which its pointer operand points.
  - Called dereferencing a pointer
- A dereferenced pointer may also be used on the left side of an assignment.

8.3 Pointer Operators (cont.)

Fig. 8.3 | Representation of `y` and `yPtr` in memory.

8.3 Pointer Operators (cont.)

Common Programming Error 8.2

Dereferencing an uninitialized pointer could cause a fatal execution-time error, or it could accidentally modify important data and allow the program to run to completion, possibly with incorrect results.

8.3 Pointer Operators (cont.)

Common Programming Error 8.3

An attempt to dereference a variable that is not a pointer is a compilation error.

8.3 Pointer Operators (cont.)

Common Programming Error 8.4

Dereferencing a null pointer is often a fatal execution-time error.

8.3 Pointer Operators (cont.)

```
// Fig. 8.4: fig08_04.cpp
// Pointer operators & and *.
#include <iostream>
using namespace std;

int main()
{
   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

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

Fig. 8.4 | Pointer operators & and *. (Part 1 of 2.)

8.3 Pointer Operators (cont.)

```
The address of a is 0012F580
The value of aPtr is 0012F580

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*&aPtr = 0012F580
```

Fig. 8.4 | Pointer operators & and *. (Part 2 of 2.)

8.3 Pointer Operators (cont.)

- The & and * operators are inverses of one another.
- Figure 8.5 lists the precedence and associativity of the operators introduced to this point.
- The address (&) and dereferencing operator (*) are unary operators on the third level.

8.4 Pass-by-Reference with Pointers

- There are three ways in C++ to pass arguments to a function—pass-by-value, pass-by-reference with reference arguments and pass-by-reference with pointer arguments.
- In this section, we explain pass-by-reference with pointer arguments.
- Pointers, like references, can be used to modify one or more variables in the caller or to pass pointers to large data objects to avoid the overhead of passing the objects by value.
- In C++, you can use pointers and the indirection operator (*) to accomplish pass-by-reference.

8.4 Pass-by-Reference with Pointers

```
// Fig. 8.6: fig08_06.cpp
// Pass-by-value used to cube a variable's value.
#include <iostream>
using namespace std;

int cubeByValue( int ); // prototype

int main()
{
   int number = 5;

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

   number = cubeByValue( number ); // pass number by value to cubeByValue
   cout << "\nThe new value of number is " << number << 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
```

Fig. 8.6 | Pass-by-value used to cube a variable's value. (Part 1 of 2.)

8.4 Pass-by-Reference with Pointers

```
The original value of number is 5
The new value of number is 125
```

Fig. 8.6 | Pass-by-value used to cube a variable's value. (Part 2 of 2.)

8.4 Pass-by-Reference with Pointers

```cpp
// Fig. 8.7: fig08_07.cpp
// Pass-by-reference with a pointer argument used to cube a
// variable's value.
#include <iostream>
using namespace std;

void cubeByReference( int * ); // prototype

int main()
{
   int number = 5;

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

   cubeByReference( &number ); // pass number address to cubeByReference

   cout << "\nThe new value of number is " << number << 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
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value. (Part 1 of 2.)

8.4 Pass-by-Reference with Pointers

```
The original value of number is 5
The new value of number is 125
```

Fig. 8.7 | Pass-by-reference with a pointer argument used to cube a variable's value. (Part 2 of 2.)

8.4 Pass-by-Reference with Pointers (cont.)

- In the function header and in the prototype for a function that expects a one-dimensional array as an argument, pointer notation may be used.
- The compiler does not differentiate between a function that receives a pointer and a function that receives a one-dimensional array.
  - The function must “know” when it’s receiving an array or simply a single variable which is being passed by reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form `int b[]`, the compiler converts the parameter to the pointer notation `int *b`.
  - Both forms are interchangeable.

8.7 `sizeof` Operator

- The unary operator `sizeof` determines the size of an array (or of any other data type, variable or constant) in bytes during program compilation.
- When applied to the name of an ar-ray, the `sizeof` operator returns the total number of bytes in the array as a value of type `size_t`.
- When applied to a pointer parameter in a function that receives an array as an argument, the `sizeof` operator returns the size of the pointer in bytes—not the size of the array.

8.7 sizeof Operator

Common Programming Error 8.7

Using the `sizeof` *operator in a function to find the size in bytes of an array parameter results in the size in bytes of a pointer, not the size in bytes of the array.*

8.7 sizeof Operator

```
// Fig. 8.14: fig08_14.cpp
// Sizeof operator when used on an array name
// returns the number of bytes in the array.
#include <iostream>
using namespace std;

size_t getSize( double * ); // prototype

int main()
{
   double array[ 20 ]; // 20 doubles; occupies 160 bytes on our system

   cout << "The number of bytes in the array is " << sizeof( array );

   cout << "\nThe number of bytes returned by getSize is "
      << getSize( array ) << endl;
} // end main

// return size of ptr
size_t getSize( double *ptr )
{
   return sizeof( ptr );
} // end function getSize
```

Fig. 8.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part 1 of 2.)

8.7 sizeof Operator

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Fig. 8.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part 2 of 2.)

8.7 sizeof Operator (cont.)

- The number of elements in an array also can be determined using the results of two `sizeof` operations.
- Consider the following array decla-ration:
  - `double realArray[ 22 ];`
- To determine the number of elements in the ar-ray, the following expression (which is evaluated at compile time) can be used:
  - `sizeof realArray / sizeof( realArray[ 0 ] )`
- The expression determines the number of bytes in array `realArray` and divides that value by the number of bytes used in memory to store the array's first element.

8.7 sizeof Operator (cont.)

```cpp
// Fig. 8.15: fig08_15.cpp
// Demonstrating the sizeof operator.
#include <iostream>
using namespace std;

int main()
{
   char c; // variable of type char
   short s; // variable of type short
   int i; // variable of type int
   long l; // variable of type long
   float f; // variable of type float
   double d; // variable of type double
   long double ld; // variable of type long double
   int array[ 20 ]; // array of int
   int *ptr = array; // variable of type int *

   cout << "sizeof c = " << sizeof c
      << "\tsizeof(char) = " << sizeof( char )
      << "\nsizeof s = " << sizeof s
      << "\tsizeof(short) = " << sizeof( short )
      << "\nsizeof i = " << sizeof i
```

Fig. 8.15 | sizeof operator used to determine standard data type sizes. (Part 1 of 2.)

8.7 sizeof Operator (cont.)

```
        << "\tsizeof(int) = " << sizeof( int )
        << "\nsizeof l = " << sizeof l
        << "\tsizeof(long) = " << sizeof( long )
        << "\nsizeof f = " << sizeof f
        << "\tsizeof(float) = " << sizeof( float )
        << "\nsizeof d = " << sizeof d
        << "\tsizeof(double) = " << sizeof( double )
        << "\nsizeof ld = " << sizeof ld
        << "\tsizeof(long double) = " << sizeof( long double )
        << "\nsizeof array = " << sizeof array
        << "\nsizeof ptr = " << sizeof ptr << endl;
} // end main
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4
```

Fig. 8.15 | `sizeof` operator used to determine standard data type sizes. (Part 2 of 2.)

8.7 `sizeof` Operator (cont.)

- Operator `sizeof` can be applied to any expression or type name.
- When `sizeof` is applied to a variable name (which is not an array name) or other expression, the num-ber of bytes used to store the specific type of the expression's value is returned.

8.8 Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- pointer arithmetic—certain arithmetic operations may be performed on pointers:
  - increment (++)
  - decremented (--)
  - an integer may be added to a pointer (+ or +=)
  - an integer may be subtracted from a pointer (- or -=)
  - one pointer may be subtracted from another of the same type

8.8 Pointer Expressions and Pointer Arithmetic

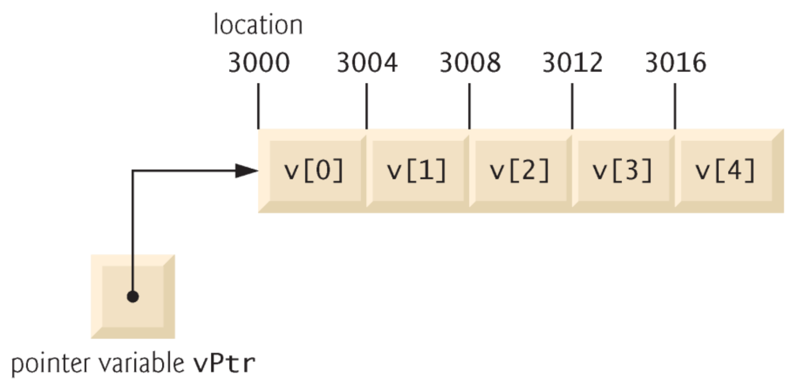

Fig. 8.16 | Array `v` and a pointer variable `int *vPtr` that points to `v`.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- Assume that array `int v[5]` has been declared and that its first element is at memory location `3000`.
- Assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is `3000`).
- Figure 8.16 diagrams this situation for a machine with four-byte integers.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- In conventional arithmetic, the addition 3000 + 2 yields the value 3002.
  – This is normally not the case with pointer arithmetic.
  – When an integer is added to, or subtracted from, a pointer, the pointer is not simply incremented or decremented by that integer, but by that integer times the size of the object to which the pointer refers.
  – The number of bytes depends on the object's data type.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

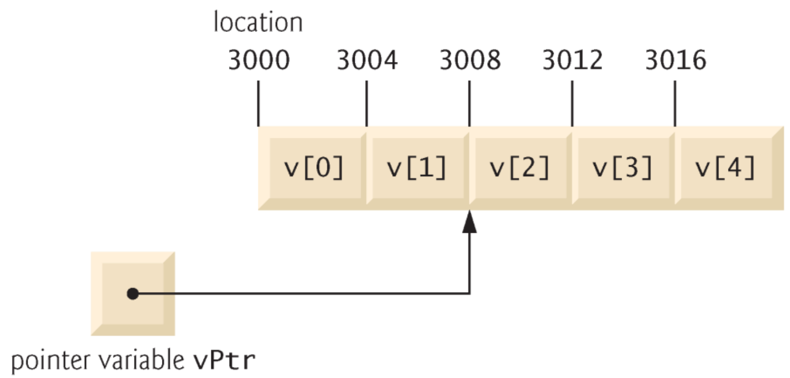

Fig. 8.17 | Pointer `vPtr` after pointer arithmetic.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- Pointer variables pointing to the same array may be subtracted from one another.
- For example, if `vPtr` contains the address `3000` and `v2Ptr` contains the address `3008`, the statement
  - `x = v2Ptr - vPtr;`
- would assign to `x` the number of array elements from `vPtr` to `v2Ptr`—in this case, `2`.
- Pointer arithmetic is meaningless unless performed on a pointer that points to an array.

8.8 Pointer Expressions and Pointer Arithmetic (cont.)

- A pointer can be assigned to another pointer if both pointers are of the same type.
- Otherwise, a cast operator (normally a `reinterpret_cast`; discussed in Section 17.8) must be used to convert the value of the pointer on the right of the assignment to the pointer type on the left of the assignment.
  – Exception to this rule is the pointer to `void` (i.e., `void *`).
- All pointer types can be assigned to a pointer of type `void *` without casting.

Questions

?