Lecture 14: Pointer – Part I

Class page: https://github.com/tsung-wei-huang/cs1410-40

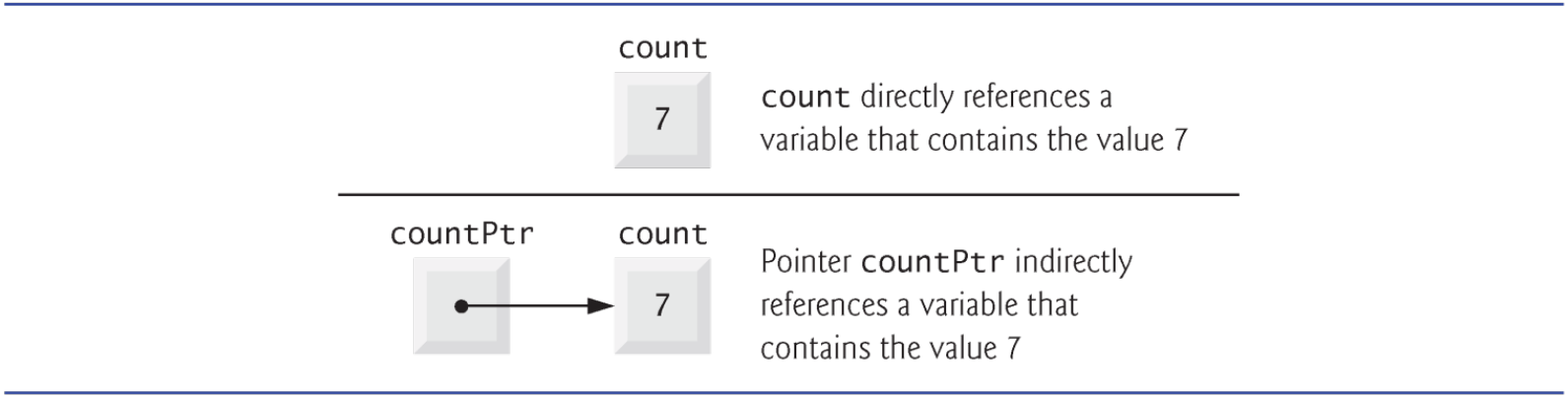
Dr. Tsung-Wei Huang
Department of Electrical and Computer Engineering
University of Utah, Salt Lake City, UT

Learning Objective

- What pointers are.
- The similarities and differences between pointers and references, and when to use each.
- To use pointers to pass arguments to functions by reference.
- The close relationships between pointers and arrays.
- To use arrays of pointers.
- Basic pointer-based string processing.
- To use pointers to functions.

Introduction

- A pointer contains the memory address of a variable.
- The variable name directly references a value, and a pointer indirectly references a value.
- Referencing a value through a pointer is called indirection.



Declaring a Pointer

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

  declares the variable `countPtr` to be of type `int *` (i.e., a pointer to an `int` value)
  - Read as "`countPtr` is a pointer to `int`."
  - Variable `count` in the preceding declaration is declared to be an `int`, not a pointer to an `int`.
  - 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.**

Value of a Pointer

- 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.
- The value 0 is the only integer value that can be assigned directly to a pointer variable.

Pointer Operator

- **The address operator (&) is a unary operator that obtains the memory address of its operand.**
  - Cannot be applied to constants or to expressions that do not result in references
- **Assuming the declarations**

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

the following statement assigns the address of the variable `y` to pointer variable `yPtr`.

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

Pointer Operator (cont'd)

- Assume the integer variable `y` stored at memory location 600000 and pointer variable `yPtr` stored at memory location 500000.
- 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
- `yPtr = 600000; *yPtr = 5;`

Example

```
1
2  // Pointer operators & and *.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int a; // a is an integer
9     int *aPtr; // aPtr is an int * which is a pointer to an integer
10
11    a = 7; // assigned 7 to a
12    aPtr = &a; // assign the address of a to aPtr
13
14    cout << "The address of a is " << &a
15       << "\nThe value of aPtr is " << aPtr;
16    cout << "\n\nThe value of a is " << a
17       << "\nThe value of *aPtr is " << *aPtr;
18    cout << "\n\nShowing that * and & are inverses of "
19       << "each other.\n&*aPtr = " << &*aPtr
20       << "\n*&aPtr = " << *&aPtr << endl;
21 } // end main
```

The & and * operators are inverses of one another.

```
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
```

Pass by Pointer (address)

- 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.
- We explain pass-by-reference with pointer arguments (pass by pointer)
- Pointers, like references, can be used to modify the variables in the caller or to pass pointers to large data objects to avoid the overhead of being passed by value.
- In C++, you can use pointers and the indirection operator (*) to accomplish pass-by-reference.

Recap: Call By Value

```
1
2  // Pass-by-value used to cube a variable's value.
3  #include <iostream>
4  using namespace std;
5
6  int cubeByValue( int ); // prototype
7
8  int main()
9  {
10    int number = 5;
11
12    cout << "The original value of number is " << number;
13
14    number = cubeByValue( number ); // pass number by value to cubeByValue
15    cout << "\nThe new value of number is " << number << endl;
16 } // end main
17
18 // calculate and return cube of integer argument
19 int cubeByValue( int n )
20 {
21    return n * n * n; // cube local variable n and return result
22 } // end function cubeByValue
```

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

How to do it with Call by Pointer?

```
1
2  // Pass-by-reference with a pointer argument used to cube a
3  // variable's value.
4  #include <iostream>
5  using namespace std;
6
7  void cubeByReference( int * ); // prototype
8
9  int main()
10 {
11    int number = 5;
12
13    cout << "The original value of number is " << number;
14
15    cubeByReference( &number ); // pass number address to cubeByReference
16
17    cout << "\nThe new value of number is " << number << endl;
18 } // end main
19
20 // calculate cube of *nPtr; modifies variable number in main
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 } // end function cubeByReference
```

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

Using const with Pointer

- **A nonconstant pointer to nonconstant data**
  ex: `int *myPtr = &x;`
  - Both the address and the data can be changed
- **A nonconstant pointer to constant data (Fig. 7.10)**
  ex: `const int *myPtr = &x;`
  - Modifiable pointer to a `const int` (data are not modifiable)
- **A constant pointer to nonconstant data (Fig. 7.11)**
  ex: `int *const myPtr = &x;`
  - Constant pointer to an `int` (data can be changed, but the address cannot)
- **A constant pointer to constant data (Fig. 7.12)**
  ex: `const int *const Ptr = &x;`
  - Both the address and the data are not modifiable

Example of const pointer

```
1
2  // Attempting to modify data through a
3  // nonconstant pointer to constant data.
4
5  void f( const int * ); // prototype
6
7  int main()
8  {
9     int y;
10
11    f( &y ); // f attempts illegal modification
12 } // end main
13
14 // xPtr cannot modify the value of constant variable to which it points
15 void f( const int *xPtr )
16 {
17    *xPtr = 100; // error: cannot modify a const object
18 } // end function f
```

Example of const pointer

```
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr is a constant pointer to a constant integer.
11    // ptr always points to the same location; the integer
12    // at that location cannot be modified.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr is const; cannot assign new value
18    ptr = &y; // error: ptr is const; cannot assign new address
19 } // end main
```

Summary

- Pointer
- Call by value
- Call by reference
- Call by address (pointer)