Pointers

Lecture 9

5.1 Introduction

- Pointers
  - Powerful, but difficult to master
  - Simulate pass-by-reference
  - Close relationship with arrays and strings

5.2 Pointer Variable Declarations and Initialization

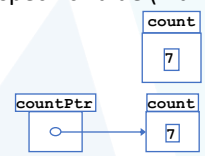
- Pointer variables
  - Contain memory addresses as values
  - Normally, variable contains specific value (direct reference)
  - Pointers contain address of variable that has specific value (indirect reference)
- Indirection
  - Referencing value through pointer
- Pointer declarations
  - ***** indicates variable is pointer

```
int *myPtr;
```

declares pointer to **int**, pointer of type **int ***

  - Multiple pointers require multiple asterisks

```
int *myPtr1, *myPtr2;
```



5.2 Pointer Variable Declarations and Initialization

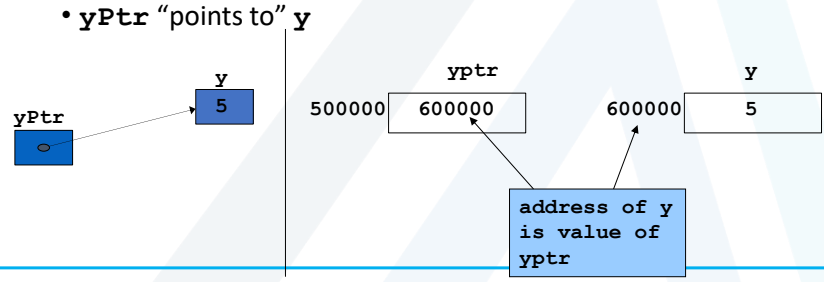
- Can declare pointers to any data type
- Pointer initialization
  - Initialized to **0**, **NULL**, or address
    - **0** or **NULL** points to nothing

5.3 Pointer Operators

- **&** (address operator)
  - Returns memory address of its operand
  - Example

```
int y = 5;
int *yPtr;
yPtr = &y;      // yPtr gets address of y
```

  - **yPtr** "points to" **y**



5.3 Pointer Operators

- ***** (indirection/dereferencing operator)
  - Returns synonym for object its pointer operand points to
  - ***yPtr** returns **y** (because **yPtr** points to **y**).
  - dereferenced pointer is lvalue

```
*yptr = 9;      // assigns 9 to y
```

- ***** and **&** are inverses of each other

fig05_04.cpp
(1 of 2)

```
1  // Fig. 5.4: fig05_04.cpp
2  // Using the & and * operators.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10   int a;     // a is an integer
11   int *aPtr;  // aPtr is a pointer to an integer
12
13   a = 7;
14   aPtr = &a;  // aPtr assigned address of a
15
16   cout << "The address of a is " << &a
17      << "\nThe value of aPtr is " << aPtr;
18
```


fig05_04.cpp
(2 of 2)

```
19   cout << "\n\nThe value of a is " << a
20      << "\nThe value of *aPtr is " << *aPtr;
21
22   cout << "\n\nShowing that * and & are inverses of "
23      << "each other.\n&*aPtr = " << &*aPtr
24      << "\n*&aPtr = " << *&aPtr << endl;
25
26   return 0;  // indicates successful termination
27
28 } // end main
```

***** and **&** are inverses of each other

fig05_04.cpp
output (1 of 1)

```
The address of a is 0012FED4
The value of aPtr is 0012FED4

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

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
*&aPtr = 0012FED4
```

***** and **&** are inverses; same result when both applied to **aPtr**

Exercise

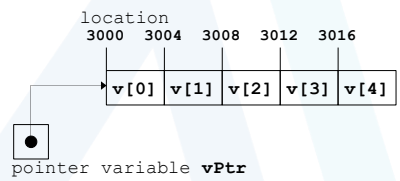
- For each of the following, write a single statement that performs the specified task. Assume that floating-point variables `number1` and `number2` have been declared and that `number1` has been initialized to 7.3.
  1. Declare the variable `fPtr` to be a pointer to an object of type `double`.
  2. Assign the address of variable `number1` to pointer variable `fPtr`.
  3. Print the value of the object pointed to by `fPtr`.
  4. Assign the value of the pointed to by `fPtr` to variable `number2`.
  5. Print the value of `number2`.
  6. Print the address of `number1`.
  7. Print the address stored in `fPtr`. Is the value printed the same as the address of `number1`?

5.7 Pointer Expressions and Pointer Arithmetic

- Pointer arithmetic
  - Increment/decrement pointer (**++** or **--**)
  - Add/subtract an integer to/from a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Pointer arithmetic meaningless unless performed on pointer to array
- 5 element **int** array on a machine using 4 byte **int**s
  - **vPtr** points to first element **v[0]**, which is at location 3000

    vPtr = 3000
  - **vPtr += 2**; sets **vPtr** to **3008**

    vPtr points to **v[2]**



5.7 Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements between two addresses

```
vPtr2 = v[ 2 ];
vPtr = v[ 0 ];
vPtr2 - vPtr == 2
```

- Pointer assignment
  - Pointer can be assigned to another pointer if both of same type
  - If not same type, cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

5.7 Pointer Expressions and Pointer Arithmetic

- Pointer comparison
  - Use equality and relational operators
  - Comparisons meaningless unless pointers point to members of same array
  - Compare addresses stored in pointers
  - Example: could show that one pointer points to higher numbered element of array than other pointer
  - Common use to determine whether pointer is 0 (does not point to anything) (ptr != NULL)

5.8 Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like constant pointer
  - Pointers can do array subscripting operations
- Accessing array elements with pointers
  - Element `b[ n ]` can be accessed by `*( bPtr + n )`
    - Called pointer/offset notation
  - Addresses
    - `&b[ 3 ]` same as `bPtr + 3`
  - Array name can be treated as pointer
    - `b[ 3 ]` same as `*( b + 3 )`
  - Pointers can be subscripted (pointer/subscript notation)
    - `bPtr[ 3 ]` same as `b[ 3 ]`

fig05_20.cpp (1 of 3)

```
1  // Fig. 5.20: fig05_20.cpp
2  // Using subscripting and pointer notations with arrays.
3
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  int main()
10 {
11   int b[] = { 10, 20, 30, 40 };
12   int *bPtr = b;   // set bPtr to point to array b
13
14   // output array b using array subscript notation
15   cout << "Array b printed with:\n"
16      << "Array subscript notation\n";
17
18   for ( int i = 0; i < 4; i++ )
19     cout << "b[" << i << "] = " << b[ i ] << '\n';
```

Using array subscript notation.

fig05_20.cpp
(2 of 3)

```
21    // output array b using the array name and
22    // pointer/offset notation
23    cout << "\nPointer/offset notation where
24       << "the pointer is the array name\n";
25
26    for ( int offset1 = 0; offset1 < 4; offset1++ )
27      cout << "*(b + " << offset1 << ") = "
28         << *( b + offset1 ) << '\n';
29
30    // output array b using bPtr and array subscript notation
31    cout << "\nPointer subscript notation\n";
32
33    for ( int j = 0; j < 4; j++ )
34      cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
35
36    cout << "\nPointer/offset notation\n";
37
```

Using array name and pointer/offset notation.

Using pointer subscript notation.

fig05_20.cpp
(3 of 3)

```
38    // output array b using bPtr and pointer/offset notation
39    for ( int offset2 = 0; offset2 < 4; offset2++ )
40      cout << "*(bPtr + " << offset2 << ") = "
41         << *( bPtr + offset2 ) << '\n';
42
43    return 0;  // indicates successful termination
44
45 } // end main
```

Using **`bPtr`** and pointer/offset notation.

fig05_20.cpp
output (1 of 2)

```
Array b printed with:

Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

fig05_20.cpp output (2 of 2)

```
 Pointer/offset notation where the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer subscript notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

fig05_21.cpp (1 of 3)

```
1  // Fig. 5.21: fig05_21.cpp
2  // Copying a string using array notation
3  // and pointer notation.
4  #include <iostream>
5
6  using std::cout;
7  using std::endl;
8
9  void copy1( char *, const char * );  // prototype
10 void copy2( char *, const char * );  // prototype
11
12 int main()
13 {
14    char string1[ 10 ];
15    char *string2 = "Hello";
16    char string3[ 10 ];
17    char string4[] = "Good Bye";
18
```

```
19    copy1( string1, string2 );
20    cout << "string1 = " << string1 << endl;
21
22    copy2( string3, string4 );
23    cout << "string3 = " << string3 << endl;
24
25    return 0;  // indicates successful termination
26
27 } // end main
28
29 // copy s2 to s1 using array notation
30 void copy1( char *s1, const char *s2 )
31 {
32    for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
33       ;  // do nothing in body
34
35 } // end function copy1
```

fig05_21.cpp (2 of 3)

Use array subscript notation to copy string in **s2** to character array **s1**.


```
37 // copy s2 to s1 using pointer notation
38 void copy2( char *s1, const char *s2 )
39 {
40    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
41       ;  // do nothing in body
42
43 } // end function copy2
```

Use pointer notation to copy string in **s2** to character array in **s1**.

Increment both pointers to point to next elements in corresponding arrays.

fig05_21.cpp output (1 of 1)

```
string1 = Hello
string3 = Good Bye
```