Computer Programming
Lecture 10

Hung-Yu Wei
Department of Electrical Engineering
National Taiwan University

7.5 Using `const` with Pointers

- `const` qualifier
  - No change is needed
    - Pointer
    - Data value
- Principle of least privilege
  - Award function enough access to accomplish task, but no more
- Four ways to pass pointer to function

  (1) Nonconstant pointer to nonconstant data

  (2) Nonconstant pointer to constant data

  (3) Constant pointer to nonconstant data

  (4) Constant pointer to constant data

4 cases

Modify Pointer? \ Modify Data Value?	Yes	No
Yes	(1)Nonconstant pointer to nonconstant data	(2)Nonconstant pointer to constant data
No	(3)Constant pointer to nonconstant data	(4)Constant pointer to constant data

(1) Nonconstant pointer to nonconstant data

- You can change both data and pointer
  - We used this previously

(2) Nonconstant pointer to constant data

- Example
  - A string with fixed character values
  - Parse the string by changing the pointer
  - Print all characters of this string
    - Moving the Ptr and display the values at each memory space
- Syntax
  - const data_type * pointer_name
  - Example

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

(3) Constant pointer to nonconstant data

- Default for array
  - Pointer to a fixed memory location (beginning of the array)
  - Values of the array elements could be changed
- Syntax
  - data_type * const pointer_name
    - Must be initialized
  - Example

```
int * const ptr = &x;
```

(4) Constant pointer to constant data

- Application scenario
  - Pass an array to a function that only reads the values (the function cannot change the values)
- Syntax
  - const data_type * const pointer_name
  - Example

```
int z=1;
const int *const ptr = &z;
```

7.6 Selection Sort

- Example using pass-by-reference
- Problem statement
  - N elements in an array
  - Sorting the array from the smallest to largest
    - Minor modification to sort from largest to smallest
- Selection sort algorithm
  - 1st iteration
    - Find the smallest among N elements
    - Swap it with the 1st element
  - 2nd iteration
    - Find the smallest among (N-1) elements (exclude the 1st one)
    - Swap it with the 2nd element
  - i-th iteration
    - Find the smallest among (N-i+1) element and swap it with the i-th element

Select and Swap

Sorting example

- Original
  - 34 56 4 10 77 51 93 30 5 52
- 1st round
  - **4** 56 **34** 10 77 51 93 30 5 52
- 2nd round
  - 4 **5** 34 10 77 51 93 30 **56** 52
- 3nd round
  - 4 5 **10** **34** 77 51 93 30 56 52
- 4-th round
  - 4 5 10 **30** 77 51 93 **34** 56 52
- …

How to program selection sort?

- Function: swap
  - Swap function is called to swap 2 elements
- For loop
  - Run N iterations
  - Each iteration find the 2 elements to swap
- Index
  - Moving the index during each iteration
  - Indicate which element to be swapped

```
1  // Fig. 7.13: fig07_13.cpp
2  // Selection sort with pass-by-reference. This program puts values into an
3  // array, sorts them into ascending order and prints the resulting array.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  void selectionSort( int * const, const int ); // prototype
9  void swap( int * const, int * const ); // prototype
10
11 int main()
12 {
13    const int arraySize = 10;
14    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16    cout << "Data items in original order\n";
17
18    for ( int i = 0; i < arraySize; i++ )
19       cout << setw( 4 ) << a[ i ];
20
21    selectionSort( a, arraySize ); // sort the array
22
23    cout << "\nData items in ascending order\n";
24
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 1 of 3.)

```
25       for ( int j = 0; j < arraySize; j++ )
26          cout << setw( 4 ) << a[ j ];
27
28       cout << endl;
29    } // end main
30
31    // function to sort an array
32    void selectionSort( int * const array, const int size )
33    {
34       int smallest; // index of smallest element
35
36       // loop over size - 1 elements
37       for ( int i = 0; i < size - 1; i++ )
38       {
39          smallest = i; // first index of remaining array
40
41          // loop to find index of smallest element
42          for ( int index = i + 1; index < size; index++ )
43
44             if ( array[ index ] < array[ smallest ] )
45                smallest = index;
46
47          swap( &array[ i ], &array[ smallest ] );
48       } // end if
49    } // end function selectionSort
```


Fig. 7.13 | Selection sort with pass-by-reference. (Part 2 of 3.)

```
50
51  // swap values at memory locations to which
52  // element1Ptr and element2Ptr point
53  void swap( int * const element1Ptr, int * const element2Ptr )
54  {
55     int hold = *element1Ptr;
56     *element1Ptr = *element2Ptr;
57     *element2Ptr = hold;
58  } // end function swap
```

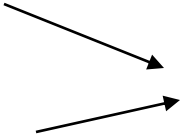
```
Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

Fig. 7.13 | Selection sort with pass-by-reference. (Part 3 of 3.)

7.7 sizeof

- sizeof
  - Determine the size (in bytes) of arrays
  - Determine the size of variables, data types, constant
- Example

```
double x[20];
cout << sizeof x;
cout << sizeof (x);
```



They are the same

  - Results:
    - 160 (bytes)
      - 8 bytes for a doulbe
      - 20 elements in the array
      - 160=8* 20

```cpp
// Fig. 7.14: fig07_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. 7.14 | `sizeof` operator when applied to an array name returns the number of bytes in the array. (Part 1 of 2.)

sizeof

- Get the number of bytes of a data type
  - sizeof (data type)
    - sizeof(int)
    - sizeof(double)
  - () is required here
  - Example (Fig 7.15)
    - Compute the number of bytes of several data types
- Count the number of elements in an array
  - double x[22];
  - sizeof x / sizeof (double);

```
1  // Fig. 7.15: fig07_15.cpp
2  // Demonstrating the sizeof operator.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char c; // variable of type char
9     short s; // variable of type short
10    int i; // variable of type int
11    long l; // variable of type long
12    float f; // variable of type float
13    double d; // variable of type double
14    long double ld; // variable of type long double
15    int array[ 20 ]; // array of int
16    int *ptr = array; // variable of type int *
17
18    cout << "sizeof c = " << sizeof c
19       << "\tsizeof(char) = " << sizeof( char )
20       << "\nsizeof s = " << sizeof s
21       << "\tsizeof(short) = " << sizeof( short )
22       << "\nsizeof i = " << sizeof i
```

Fig. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part 1 of 2.)

```
23          << "\tsizeof(int) = " << sizeof( int )
24          << "\nsizeof l = " << sizeof l
25          << "\tsizeof(long) = " << sizeof( long )
26          << "\nsizeof f = " << sizeof f
27          << "\tsizeof(float) = " << sizeof( float )
28          << "\nsizeof d = " << sizeof d
29          << "\tsizeof(double) = " << sizeof( double )
30          << "\nsizeof ld = " << sizeof ld
31          << "\tsizeof(long double) = " << sizeof( long double )
32          << "\nsizeof array = " << sizeof array
33          << "\nsizeof ptr = " << sizeof ptr << endl;
34    } // 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. 7.15 | `sizeof` operator used to determine standard data type sizes. (Part 2 of 2.)

7.8 Pointer Expressions and Pointer Arithmetic

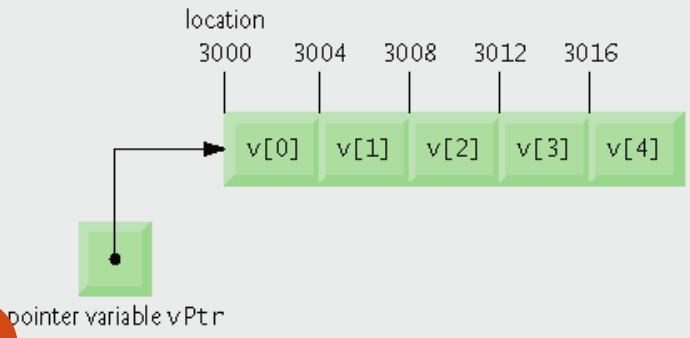
- Pointer arithmetic
  - Increment/decrement pointer (**++** or **--**)
  - Add/subtract an integer to/from a pointer(**+** or **+=** , **-** or **-=**)
  - Pointers may be subtracted from each other
  - Pointer arithmetic meaningless unless performed on pointer to array
- Initialization
  - int *vPtr=v; // **v** is the array name
  - int *vPtr=&v[0];

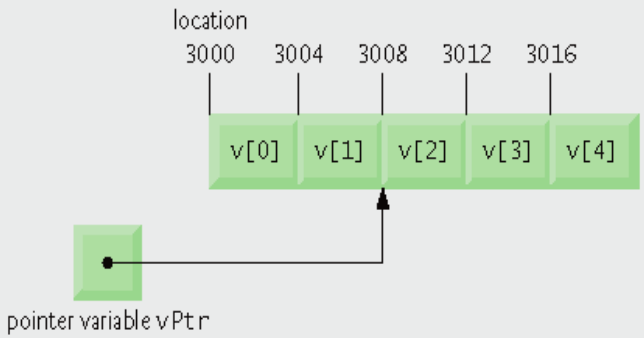
Example

- 5 element int array on a machine using 4 byte ints
  - 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]
- ++vPtr and vPtr++
  - Move to the next element
- --vPtr and vPt--
  - Move to the previous element





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 the 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

7.9 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]

Example:b[1], *(b+1), bPtr[1], *(bPtr+1)

```
1  // Fig. 7.18: fig07_18.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; // create 4-element array b
9     int *bPtr = b; // set bPtr to point to array b
10
11    // output array b using array subscript notation
12    cout << "Array b printed with:\n\nArray subscript notation\n";
13
14    for ( int i = 0; i < 4; i++ )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17    // output array b using the array name and pointer/offset notation
18    cout << "\nPointer/offset notation where "
19       << "the pointer is the array name\n";
20
21    for ( int offset1 = 0; offset1 < 4; offset1++ )
22       cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
```

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 1 of 3.)

```
23
24      // output array b using bPtr and array subscript notation
25      cout << "\nPointer subscript notation\n";
26
27      for ( int j = 0; j < 4; j++ )
28         cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
29
30      cout << "\nPointer/offset notation\n";
31
32      // output array b using bPtr and pointer/offset notation
33      for ( int offset2 = 0; offset2 < 4; offset2++ )
34         cout << "*(bPtr + " << offset2 << ") = "
35            << *( bPtr + offset2 ) << '\n';
36   } // end main
```

```
Array b printed with:

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

Fig. 7.18 | Referencing array elements with the array name and with pointers. (Part 2 of 3.)

7.10 Pointer-Based String Processing

- A pointer-based string
  - An array of characters ending in the '\0' null character
  - A string is accessed via a pointer to its 1st character
- Declaration
  - char color[]="blue";
  - const char *colorPtr="blue";
  - char color[]={'b','l','u','e','\0'};

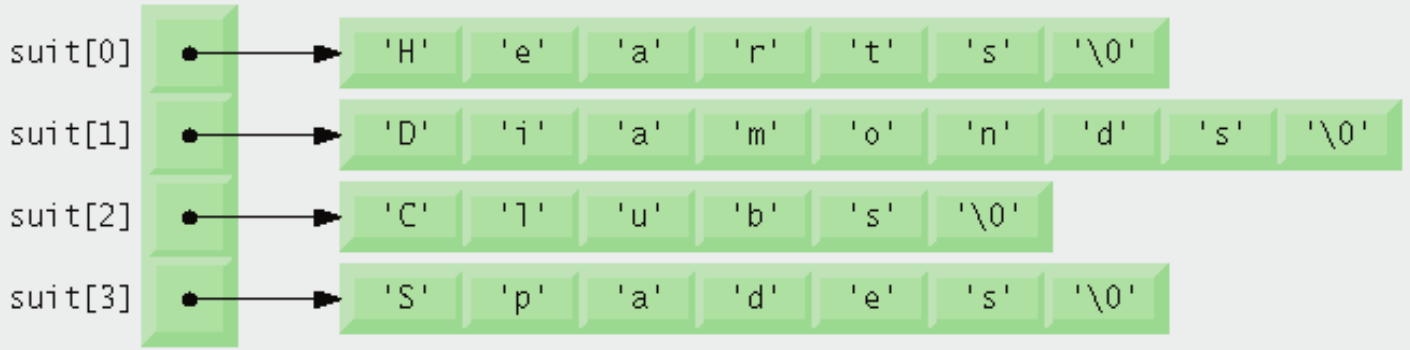
getline to a character string

- cin.getline
  - <iostream>
  - Syntax
    - cin.getline(array, size, delimiter);
- Read line of text
- Copies input into specified **`array`** until either
  - One less than **`size`** is reached
  - **`delimiter`** character is input
- Example

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

7.11 Array of pointers

- Array of pointers
  - Form an array of pointer-based string
  - Also known as *string array*
- Example
  - const char *suit[4] ={“Hearts”, “Diamonds”, “Clubs”,”Spades”}



7.12 Function Pointers

- Pointers to functions
  - Contain address of function
  - Similar to how array name is address of first element
  - Function that defines function
- Example
  - bool (*compare) (int,int)
    - Pointer to a function named *compare*
    - Need ()
    - name is the starting address of code

Example codes

- Use function pointer to implement multipurpose selection sort
- Selection sort
  - Ascending
  - Descending
- Two functions ("*ascending*" and "*descending*") determine the order of sorting
  - Pass a pointer to function "ascending"
  - Or pass a pointer to function "descending"

```
1  // Fig. 7.20: fig07_20.cpp
2  // Multipurpose sorting program using function pointers.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  // prototypes
8  void selectionSort( int [], const int, bool (*)( int, int ) );
9  void swap( int * const, int * const );
10 bool ascending( int, int ); // implements ascending order
11 bool descending( int, int ); // implements descending order
12
13 int main()
14 {
15    const int arraySize = 10;
16    int order; // 1 = ascending, 2 = descending
17    int counter; // array index
18    int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
19
20    cout << "Enter 1 to sort in ascending order,\n"
21       << "Enter 2 to sort in descending order: ";
22    cin >> order;
23    cout << "\nData items in original order\n";
24
```


Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 1 of 5.)

```
25       // output original array
26       for ( counter = 0; counter < arraySize; counter++ )
27          cout << setw( 4 ) << a[ counter ];
28
29       // sort array in ascending order; pass function ascending
30       // as an argument to specify ascending sorting order
31       if ( order == 1 )
32       {
33          selectionSort( a, arraySize, ascending );
34          cout << "\nData items in ascending order\n";
35       } // end if
36
37       // sort array in descending order; pass function descending
38       // as an argument to specify descending sorting order
39       else
40       {
41          selectionSort( a, arraySize, descending );
42          cout << "\nData items in descending order\n";
43       } // end else part of if...else
44
45       // output sorted array
46       for ( counter = 0; counter < arraySize; counter++ )
47          cout << setw( 4 ) << a[ counter ];
48
```


Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 2 of 5.)

```
49       cout << endl;
50    } // end main
51
52    // multipurpose selection sort; the parameter compare is a pointer to
53    // the comparison function that determines the sorting order
54    void selectionSort( int work[], const int size,
55                        bool (*compare)( int, int ) )
56    {
57       int smallestOrLargest; // index of smallest (or largest) element
58
59       // loop over size - 1 elements
60       for ( int i = 0; i < size - 1; i++ )
61       {
62          smallestOrLargest = i; // first index of remaining vector
63
64          // loop to find index of smallest (or largest) element
65          for ( int index = i + 1; index < size; index++ )
66             if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
67                smallestOrLargest = index;
68
69          swap( &work[ smallestOrLargest ], &work[ i ] );
70       } // end if
71    } // end function selectionSort
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 3 of 5.)

```
72
73  // swap values at memory locations to which
74  // element1Ptr and element2Ptr point
75  void swap( int * const element1Ptr, int * const element2Ptr )
76  {
77     int hold = *element1Ptr;
78     *element1Ptr = *element2Ptr;
79     *element2Ptr = hold;
80  } // end function swap
81
82  // determine whether element a is less than
83  // element b for an ascending order sort
84  bool ascending( int a, int b )
85  {
86     return a < b; // returns true if a is less than b
87  } // end function ascending
88
89  // determine whether element a is greater than
90  // element b for a descending order sort
91  bool descending( int a, int b )
92  {
93     return a > b; // returns true if a is greater than b
94  } // end function descending
```

Fig. 7.20 | Multipurpose sorting program using function pointers. (Part 4 of 5.)

Summary: Pointers

- Basic pointers
  - Declaration, Initialization, Operations
- Pointers and arrays
- Pointers and strings
- Pointers and functions
  - Pass-by-reference
- Other topics
  - const
  - sizeof
  - function pointer
    - Example: Selection Sort