Lab # 16: Arrays and Strings – Part 2

EC-102 – Computer Systems and Programming

Usman Ayub Sheikh

School of Mechanical and Manufacturing Engineering (SMME),
National University of Sciences and Technology (NUST)

Monday 21st December, 2015

Outline

Passing Arrays to Functions

```
1  #include <iostream>
2  #include <iomanip>
3  using namespace std;
4
5  const int DISTRICTS = 4;
6  const int MONTHS = 3;
7  void display(double[DISTRICTS][MONTHS]); // declaration
8
9  int main()
10 {
11     double sales[DISTRICTS][MONTHS] = {
12         {1432.07, 234.50, 654.01},
13         {322.00, 13838.32, 17589.88},
14         {9328.34, 934.00, 4492.30},
15         {12838.29, 2332.63, 32.93}};
16     display(sales); // call
17     cout << endl;
18
19     return 0;
20 }
```

Passing Arrays to Functions

```
21 // definition
22 void display(double funsales[DISTRICTS][MONTHS])
23 {
24     int d, m;
25     cout << "\n\n";
26     cout << "                              Month\n";
27     cout << "                    1          2          3";
28     for (d = 0; d < DISTRICTS; d++)
29     {
30         cout << "\nDistrict " << d + 1;
31         for (m = 0; m < MONTHS; m++)
32         {
33             cout << setw(10) << funsales[d][m];
34         }
35     }
36 }
```

Passing Arrays to Functions

Function Declaration:
Array arguments are represented by the **data type** and **size**

```
void display(double[DISTRICTS][MONTHS]);
```

Function Call:
Only the **name** of the array is used as an argument

```
display(sales);
```

Function Definition:
In function definition, the declarator looks like this:

```
void display(double funsales[DISTRICTS][MONTHS])
```

The array argument uses the **data type**, a **name** and the **sizes** of the array dimensions

Arrays of Structures

Arrays can contain structures as well as simple data types.

```
1  #include <iostream>
2  using namespace std;
3
4  struct Part
5  {
6      int modelnumber;
7      int partnumber;
8      float cost;
9  };
10
11 int main()
12 {
13     const int SIZE = 4;
14     int n;
15
16     Part apart[SIZE]; // define array of structures
```

Arrays of Structures

```
17 for (int n = 0; n < SIZE; n++) // get values for all
18     {
19         cout << endl;
20         cout << "Enter model number: "; // get model number
21         cin >> apart[n].modelnumber;
22 
23         cout << "Enter part number: "; // get part number
24         cin >> apart[n].partnumber;
25 
26         cout << "Enter cost: "; // get cost
27         cin >> apart[n].cost;
28     }
29     cout << endl;
```

Arrays of Structures

```
30 for (int n = 0; n < SIZE; n++) // show values for all
31     {
32         cout << "Model " << apart[n].modelnumber;
33         cout << " Part " << apart[n].partnumber;
34         cout << " Cost " << apart[n].cost << endl;
35     }
36
37     return 0;
38 }
```

Strings

- Array of type char
- As with other data types, strings can be variables or constants.
- The following example asks the user to enter a string and places this string in a string variable

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      const int MAX = 80;
7      char str[MAX];
8
9      cout << "Enter a string: "; cin >> str;
10     cout << "You entered: " << str << endl;
11     return 0;
12 }
```

Avoiding Buffer Overflow

- What happens if a user enters a string that is longer than the array used to hold it?
  This overly enthusiastic typist would end up crashing the system.
- It is possible to tell the >> operator to limit the number of characters it places in an array
- One way to do that is to use a `setw` manipulator to specify the maximum number of characters the input buffer can accept

Avoiding Buffer Overflow

```
1  // avoiding buffer overflow using setw manipulator
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  int main()
7  {
8      const int MAX = 20;
9      char str[MAX];
10
11     cout << "Enter a string: "; cin >> setw(MAX) >> str;
12     cout << "You entered: " << str << endl;
13     return 0;
14 }
```

String Constants

You can initialize a string to a constant value when you define it

```
1 // initialized string
2 #include <iostream>
3 using namespace std;
4 int main()
5 {
6     char str[] = "This string has been initialized";
7     cout << str << endl;
8     return 0;
9 }
```

Reading Blanks

- The extraction operator (>>) considers the <space> to be a terminating character
- So, for the following program, if a user enters "Hello, there", `cout << str;` will print "`Hello,`" only

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      char str[20];
7      cout << "Enter a string: "; cin >> str;
8      cout << str << endl;
9      return 0;
10 }
```

- Anything typed after a space is thrown away
- This problem can be solved by using another function `cin.get()`

Reading Blanks

```
1  // reads string with blanks
2  #include <iostream>
3  using namespace std;
4  int main()
5  {
6      const int MAX = 80;
7      char str[MAX];
8
9
10     cout << "\nEnter a string: ";
11     cin.get(str, MAX);
12     cout << "You entered: " << str << endl;
13     return 0;
14 }
```

Reading Multiple Lines

- The third argument of `cin.get()` specifies the character that tells the function to stop reading
- The default value of this argument is the newline
- The following example overrides the default value by '$'

```
1  // reads multiple lines, terminates on \$ character
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      const int MAX = 2000;
8      char str[MAX];
9      cout << "\nEnter a string:\n";
10
11     cin.get(str, MAX, '$'); // terminate with $
12     cout << "You entered:\n" << str << endl;
13     return 0;
14 }
```