Lecture 13: Functions & Arrays

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 19th, 2010
Two ways to pass arguments to functions in many programming languages are pass-by-value and pass-by-reference.

When an argument is passed by value, a *copy of the argument’s value is made and passed* (on the function call stack) to the called function.

- Changes to the copy do not affect the original variable’s value in the caller.

To specify a reference to a constant, place the `const` qualifier before the type specifier in the parameter declaration.
Performance Tip 6.5

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.
6.14 References and Reference Parameters (cont.)

• With pass-by-reference, the caller gives the called function the ability to access the caller’s data directly, and to modify that data.

• A reference parameter is an alias for its corresponding argument in a function call.

• To indicate that a function parameter is passed by reference, simply follow the parameter’s type in the function prototype by an ampersand (&); use the same convention when listing the parameter’s type in the function header.
Performance Tip 6.6

Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.
Software Engineering Observation 6.12
Pass-by-reference can weaken security; the called function can corrupt the caller’s data.
**Common Programming Error 6.11**

Because reference parameters are mentioned only by name in the body of the called function, you might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original variables are changed by the function.
// Fig. 6.19: fig06_19.cpp
// Comparing pass-by-value and pass-by-reference with references.
#include <iostream>
using namespace std;

int squareByValue(int); // function prototype (value pass)
void squareByReference(int &); // function prototype (reference pass)

int main()
{
    int x = 2; // value to square using squareByValue
    int z = 4; // value to square using squareByReference

    // demonstrate squareByValue
    cout << "x = " << x << " before squareByValue\n";
    cout << "Value returned by squareByValue: "
         << squareByValue(x) << endl;
    cout << "x = " << x << " after squareByValue\n" << endl;

    // demonstrate squareByReference
    cout << "z = " << z << " before squareByReference" << endl;
    squareByReference(z);
    cout << "z = " << z << " after squareByReference" << endl;
} // end main
6.14 References and Reference Parameters (cont.)

```java
25 // squareByValue multiplies number by itself, stores the
26 // result in number and returns the new value of number
27 int squareByValue(int number)
28 {
29     return number *= number; // caller's argument not modified
30 } // end function squareByValue

33 // squareByReference multiplies numberRef by itself and stores the result
34 // in the variable to which numberRef refers in function main
35 void squareByReference(int &numberRef)
36 {
37     numberRef *= numberRef; // caller's argument modified
38 } // end function squareByReference
```

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference

**Fig. 6.19** Passing arguments by value and by reference. (Part 2 of 2.)
Performance Tip 6.7

For passing large objects, use a constant reference parameter to simulate the appearance and security of pass-by-value and avoid the overhead of passing a copy of the large object.
6.14 References and Reference Parameters (cont.)

• References can also be used as aliases for other variables within a function.
• Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables.
• Once a reference is declared as an alias for another variable, all operations supposedly performed on the alias are actually performed on the original variable.
// Fig. 6.20: fig06_20.cpp

// Initializing and using a reference.

#include <iostream>

using namespace std;

int main()
{
    int x = 3;
    int &y = x; // y refers to (is an alias for) x
    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7; // actually modifies x
    cout << "x = " << x << endl << "y = " << y << endl;
}

x = 3
y = 3
x = 7
y = 7

Fig. 6.20  |  Initializing and using a reference.
```cpp
// Fig. 6.21: fig06_21.cpp
// References must be initialized.
#include <iostream>
using namespace std;

int main()
{
    int x = 3;
    int &y; // Error: y must be initialized
    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7;
    cout << "x = " << x << endl << "y = " << y << endl;
} // end main
```

Microsoft Visual C++ compiler error message:

C:\cpphttp7_examples\ch06\Fig06_21\fig06_21.cpp(9) : error C2530: 'y' : references must be initialized

GNU C++ compiler error message:

fig06_21.cpp:9: error: 'y' declared as a reference but not initialized

**Fig. 6.21** | Uninitialized reference causes a syntax error.
Functions can return references, but this can be dangerous.

When return-ing a reference to a variable declared in the called function, the variable should be declared `static` in that function.
Common Programming Error 6.12

Returning a reference to an automatic variable in a called function is a logic error. Some compilers issue a warning when this occurs.
It’s possible to declare local and global variables of the same name.

C++ provides the unary scope resolution operator (::) to access a global variable when a local variable of the same name is in scope.

Using the unary scope resolution operator (::) with a given variable name is optional when the only variable with that name is a global variable.
6.16 Unary Scope Resolution Operator

```cpp
// Fig. 6.23: fig06_23.cpp
// Using the unary scope resolution operator.
#include <iostream>
using namespace std;

int number = 7; // global variable named number

int main()
{
    double number = 10.5; // local variable named number
    // display values of local and global variables
    cout << "Local double value of number = " << number
        << "\nGlobal int value of number = " << ::number << endl;
} // end main
```

Fig. 6.23

Local double value of number = 10.5
Global int value of number = 7

variable name is optional when the only variable with that name is a global variable.
Error-Prevention Tip 6.4

Avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors.
C++ enables several functions of the same name to be defined, as long as they have different signatures.

This is called **function overloading**.

The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call.

Function overloading is used to create several functions of the same name that perform similar tasks, but on different data types.
```cpp
// Fig. 6.24: fig06_24.cpp
// Overloaded functions.
#include <iostream>
using namespace std;

// function square for int values
int square( int x )
{
    cout << "square of integer " << x << " is ";
    return x * x;
} // end function square with int argument

// function square for double values
double square( double y )
{
    cout << "square of double " << y << " is ";
    return y * y;
} // end function square with double argument
```

**Fig. 6.24** Overloaded square functions. (Part I of 2.)
```cpp
20 int main()
21 {
22    cout << square( 7 ); // calls int version
23    cout << endl;
24    cout << square( 7.5 ); // calls double version
25    cout << endl;
26 } // end main
```

Fig. 6.24 Overloaded square functions. (Part 2 of 2.)
• Overloaded functions are distinguished by their signatures.

• A signature is a combination of a function’s name and its parameter types (in order).
This chapter introduces the important topic of data structures—collections of related data items.

Arrays are data structures consisting of related data items of the same type.

After discussing how arrays are declared, created and initialized, we present a series of practical examples that demonstrate several common array manipulations.
An array is a consecutive group of memory locations that all have the same type.

To refer to a particular location or element in the array, specify the name of the array and the position number of the particular element.

Figure 7.1 shows an integer array called c.

12 elements.

The position number is more formally called a subscript or index (this number specifies the number of elements from the beginning of the array).

The first element in every array has subscript 0 (zero) and is sometimes called the zeroth element.

The highest subscript in array c is 11, which is 1 less than the number of elements in the array (12).

A subscript must be an integer or integer expression (using any integral type).
7.2 Arrays

Fig. 7.1 | Array of 12 elements.
Common Programming Error 7.1

Note the difference between the "seventh element of the array" and "array element 7." Array subscripts begin at 0, so the "seventh element of the array" has a subscript of 6, while "array element 7" has a subscript of 7 and is actually the eighth element of the array. Unfortunately, this distinction frequently is a source of off-by-one errors. To avoid such errors, we refer to specific array elements explicitly by their array name and subscript number (e.g., c[6] or c[7]).
## 7.2 Arrays

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td>left to right</td>
<td>scope resolution</td>
</tr>
<tr>
<td>( ) [ ]</td>
<td>left to right</td>
<td>highest</td>
</tr>
<tr>
<td>++ -- static_cast&lt; type &gt;( operand )</td>
<td>left to right</td>
<td>unary (postfix)</td>
</tr>
<tr>
<td>++ -- + - !</td>
<td>right to left</td>
<td>unary (prefix)</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>left to right</td>
<td>insertion/extraction</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right to left</td>
<td>conditional</td>
</tr>
<tr>
<td>= += -= *= /= %=</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
<td>comma</td>
</tr>
</tbody>
</table>

**Fig. 7.2** | Operator precedence and associativity.
• Arrays occupy space in memory.

• To specify the type of the elements and the number of elements required by an array use a declaration of the form:

  ```c
  type arrayName[arraySize];
  ```

• The compiler reserves the appropriate amount of memory.

• Arrays can be declared to contain values of any nonreference data type.
7.4.1 Declaring an Array and Using a Loop to Initialize the Array’s Elements

```cpp
// Fig. 7.3: fig07_03.cpp
// Initializing an array.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    int n[10]; // n is an array of 10 integers

    // initialize elements of array n to 0
    for (int i = 0; i < 10; i++)
        n[i] = 0; // set element at location i to 0

    cout << "Element" << setw(13) << "Value" << endl;

    // output each array element's value
    for (int j = 0; j < 10; j++)
        cout << setw(7) << j << setw(13) << n[j] << endl;
} // end main
```

**Fig. 7.3** Initializing an array’s elements to zeros and printing the array. (Part 1 of 2.)
### Declaring an Array and Using a Loop to Initialize the Array’s Elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

**Fig. 7.3** | Initializing an array’s elements to zeros and printing the array. (Part 2 of 2.)
The elements of an array also can be initialized in the array declaration by following the array name with an equals sign and a brace-delimited comma-separated list of initializers.

The program in Fig. 7.4 uses an initializer list to initialize an integer array with 10 values (line 10) and prints the array in tabular format (lines 12–16).

If there are fewer initializers than elements in the array, the remaining array elements are initialized to zero.

If the array size is omitted from a declaration with an initializer list, the compiler determines the number of elements in the array by counting the number of elements in the initializer list.
7.4.2 Initializing an Array in a Declaration with an Initializer List

```cpp
// Fig. 7.4: fig07_04.cpp
// Initializing an array in a declaration.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // use initializer list to initialize array n
    int n[10] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };

    cout << "Element" << setw(13) << "Value" << endl;

    // output each array element's value
    for (int i = 0; i < 10; i++)
    {
        cout << setw(7) << i << setw(13) << n[i] << endl;
    } // end main
```

**Fig. 7.4** Initializing the elements of an array in its declaration. (Part 1 of 2.)
### 7.4.2 Initializing an Array in a Declaration with an Initializer List

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>1</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>95</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>90</td>
</tr>
<tr>
<td>7</td>
<td>70</td>
</tr>
<tr>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>9</td>
<td>37</td>
</tr>
</tbody>
</table>

**Fig. 7.4** | Initializing the elements of an array in its declaration. (Part 2 of 2.)
Figure 7.5 sets the elements of a 10-element array \( s \) to the even integers 2, 4, 6, …, 20 (lines 14–15) and prints the array in tabular format (lines 17–21).

Line 10 uses the `const` qualifier to declare a so-called constant variable `arraySize` with the value 10.

Constant variables must be initialized with a constant expression when they’re declared and cannot be modified thereafter.

Constant variables are also called named constants or read-only variables.
// Fig. 7.5: fig07_05.cpp
// Set array s to the even integers from 2 to 20.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // constant variable can be used to specify array size
    const int arraySize = 10;

    int s[ arraySize ]; // array s has 10 elements

    for ( int i = 0; i < arraySize; i++ ) // set the values
        s[ i ] = 2 + 2 * i;

    cout << "Element" << setw( 13 ) << "Value" << endl;

    // output contents of array s in tabular format
    for ( int j = 0; j < arraySize; j++ )
        cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
}

Fig. 7.5 | Generating values to be placed into elements of an array. (Part 1 of 2.)
### 7.4.3 Specifying an Array’s Size with a Constant Variable and Setting Array Elements with Calculations

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>20</td>
</tr>
</tbody>
</table>

**Fig. 7.5** Generating values to be placed into elements of an array. (Part 2 of 2.)
7.4.3 Specifying an Array’s Size with a Constant Variable and Setting Array Elements with Calculations

```c++
// Fig. 7.6: fig07_06.cpp
// Using a properly initialized constant variable.
#include <iostream>
using namespace std;

int main()
{
    const int x = 7; // initialized constant variable
    cout << "The value of constant variable x is: " << x << endl;
} // end main
```

The value of constant variable x is: 7

**Fig. 7.6** Initializing and using a constant variable.
```cpp
// Fig. 7.7: fig07_07.cpp
// A const variable must be initialized.

int main()
{
    const int x; // Error: x must be initialized
    x = 7; // Error: cannot modify a const variable
}
// end main

Microsoft Visual C++ compiler error message:
C:\cpphtp7_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object must be initialized if not extern
C:\cpphtp7_examples\ch07\fig07_07.cpp(8) : error C3892: 'x' : you cannot assign to a variable that is const

GNU C++ compiler error message:
fig07_07.cpp:6: error: uninitialized const 'x'
fig07_07.cpp:8: error: assignment of read-only variable 'x'

Fig. 7.7 | const variables must be initialized.
```
Common Programming Error 7.5

Only constants can be used to declare the size of automatic and static arrays. Not using a constant for this purpose is a compilation error.
Good Programming Practice 7.2

Defining the size of an array as a constant variable instead of a literal constant makes programs clearer. This technique eliminates so-called magic numbers. For example, repeatedly mentioning the size 10 in array-processing code for a 10-element array gives the number 10 an artificial significance and can be confusing when the program includes other 10s that have nothing to do with the array size.
7.4.4 Summing the Elements of an Array

- Often, the elements of an array represent a series of values to be used in a calculation.
- The program in Fig. 7.8 sums the values contained in the 10-element integer array \( a \).
7.4.4 Summing the Elements of an Array

```cpp
// Fig. 7.8: fig07_08.cpp
// Compute the sum of the elements of the array.
#include <iostream>
using namespace std;

int main()
{
    const int arraySize = 10; // constant variable indicating size of array
    int a[ arraySize ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
    int total = 0;

    // sum contents of array a
    for ( int i = 0; i < arraySize; i++ )
        total += a[ i ];

    cout << "Total of array elements: " << total << endl;
} // end main
```

Total of array elements: 849

**Fig. 7.8** | Computing the sum of the elements of an array.
Questions