

Lecture 5

C++ Programming

Arne Kutzner

Hanyang University / Seoul Korea

Reference Variables, Reference Parameter

Reference Variables

- A *reference* variable is an alternative name for a variable. A *shortcut*.
- **A reference variable must be initialized to *reference* another variable.**
- Once the reference is initialized you can treat it just like any other variable.
- Info in the Web:
<https://en.cppreference.com/w/cpp/language/reference>

Reference Variable Declarations

- To declare a reference variable you precede the variable name with a “&”:

```
Int x; double d; char ch;
```

```
    int &foo = x;
```

```
    double &blah = d;
```

```
    char &c = ch;
```

Reference Variable Example

```
int count;  
int &blah = count;  
// blah is the same variable as count  
  
count = 1;  
cout << "blah is " << blah << endl;  
blah++;  
cout << "count is " << count << endl;
```

Call-by-value vs. Call-by-reference

- So far we looked at functions that get a copy of what the *caller* passed in.
 - This is call-by-value, as the value is what gets passed in (the value of a variable).
- We can also define functions that are passed a *reference* to a variable.
 - This is call-by-reference, the function can change a callers variables directly.

Reference Parameters


- As for variables, you can declare reference parameters:

```
void add10( int &x) {  
    x = x+10;  
}
```

...

```
add10(counter);
```

The parameter is a reference



Swap function using reference variables

With reference variables, :


```
void swap( int &x, int &y) {  
    int tmp;  
    tmp = x;  
    x = y;  
    y = tmp;  
}
```


References plus **const**

- You can use **const** in order to prohibit the change of the value of some reference variable

```
void wrong(const int &x) {  
    x = 2;  
}
```

here you get a compile error



- However, **const** plus reference allows the passing of values as well.

Arrays

Arrays - Introduction

- Simple data types, that we have seen so far, use a single memory cell to store a single value.
- An array is a collection of adjacent/consecutive memory cells of equal type that are associated with a single symbolic name.
- Individual elements of an array are accessed using an index (number)

Arrays - Declaration

- We must declare:
 1. The datatype of the array elements
 2. the name of the array
 3. the number of data items (i.e. elements)
- Example: `double x[8];`



**datatype of all
array
elements**

name of array

**number of
elements**

Access of array elements

- To access an element of some array we specify:
 1. The name of the array
 2. The number of the element, that we would like to read/write. This number is called subscript

x[i]

Array name **subscript**

Arrays indexing start at 0 !!!!!!!

- The first element is the 0th element!
- If you declare an array of n elements, the last one is number $n-1$.
- If you try to access element number n it is an error!
- A subscript can be any integer expression:
These are all valid subscripts:
`x[17]` **`x[i+3]`** **`x[a+b+c]`**

Memory-Picture of an array

each **double** is 8 bytes

8 bytes

`double x[8];`

`x[0]`

`x[1]`

`x[7]`

Note that the last element of the array is referred by `x[7]` not `x[8]`

Examples of statements for manipulating some array

Statement	Explanation
<code>std::cout << x[0];</code>	Prints the value of x[0]
<code>x[3] = 25.0;</code>	Stores the value 25.0 in x[3]
<code>sum = x[0] + x[1];</code>	Stores the sum of x[0] and x[1] in the variable sum
<code>sum += x[2];</code>	Adds the value of x[2] to sum
<code>x[3] += 1.0;</code>	Adds 1.0 to x[3]
<code>x[2] = x[0] + x[1];</code>	Stores the sum of x[0] and x[1] in x[2];



Invalid Subscript Reference

- To create a valid reference, the value of the array subscript must lie between 0 and one less than the declared size of the array.

```
int x[8];
```

```
...
```

```
...
```

```
std::cout << x[10];
```

C++ Programming

**Invalid
reference!
Run-time error
or incorrect
result**

Array Initialization

- Recall that a simple variable can be initialized at the same time when it is declared.

```
int sum = 0;
```

- Likewise, arrays can also be initialized in declaration:

In this case, you don't have to provide the number of array elements



```
double x[]={16.0,12.0,6.0,8.0,2.5,12.0,14.0,-54.5};
```

- Another example:

```
char vowels[]={ 'A' , 'E' , 'I' , 'O' , 'U' };
```

Arrays of `char` are special

- C++ provides a special way to deal with arrays of characters:

```
char s[] = "Some text";
```

- `char` arrays can be initialized with string literals.
- These strings are NULL-terminated like C-strings
- There is a class based implementation for strings as well.

Arrays and Functions

Arrays and Functions - Introduction

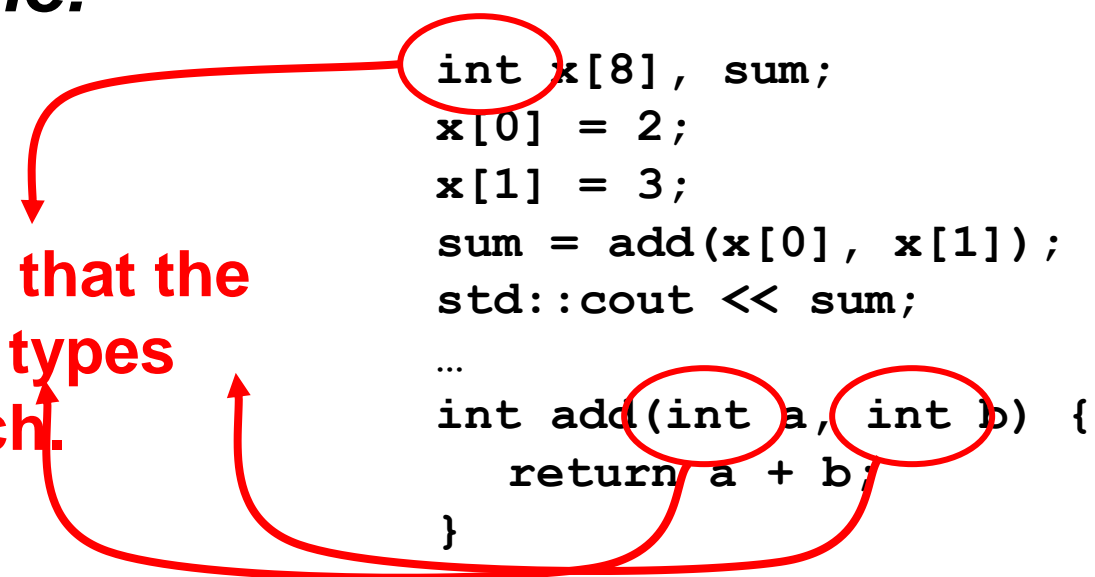
- There are two ways of using arrays as function arguments:
 - **Use the individual array element:** the function can use only those array elements that are passed into the function.
 - **Use of the entire array:** the function has access to all array elements and can manipulate them.

Passing of single array elements

- Array elements when passed into a function are treated as if they were simple variables of the underlying data types.

Example:

Note that the data types match.



```
int x[8], sum;  
x[0] = 2;  
x[1] = 3;  
sum = add(x[0], x[1]);  
std::cout << sum;  
...  
int add(int a, int b) {  
    return a + b;  
}
```

The diagram illustrates the data type matching between array elements and function parameters. Red circles highlight the `int` type in the function signature `int add(int a, int b)` and the `int` type in the array declaration `int x[8]`. Red arrows point from the `int` in the function signature to the `int` in the array declaration, and from the `int` in the function signature to the `int` in the array element access `x[0]` and `x[1]`, indicating that the data types match.

Passing of complete Arrays

- Passing the complete array as argument:

Example:

```
double y[10];  
fill_array(y, 10, 0.0);  
...  
void fill_array(double x[], int n, double in_value) {  
    /* sets all elements of the array to in_value */  
    int i;  
    for (i = 0; i < n; i++)  
        x[i]=in_value;  
}
```

**the entire array y is
passed into the function**

**the number of elements is
not specified**

**If the function needs to
know the array size, this
should be passed as
function argument.**

Call by Reference

- When an array is passed entirely to a function this is done by using “**call by reference**”.
- Call by reference means:
All changes to some array **A** inside some function are although effective wherever we later use the array **A** outside this function

Example for use of call by reference

- At the end **sum** contains all sums

```
double x[10], y[10], sum[10];  
fill_array(x, 10, 2.0);  
fill_array(y, 10, 3.0);  
add_arrays(x, y, sum, 10);  
...  
void add_arrays(double x[], double y[], double sum[], int n)  
{  
    int i;  
    for (i=0; i < n; i++)  
        sum[i] = x[i] + y[i];  
}
```

The diagram illustrates call by reference. In the `add_arrays` function call, the arguments `x`, `y`, and `sum` are circled in red. A red arrow points from the circled `x` and `y` to the text "input arrays". Another red arrow points from the circled `sum` to the text "output array".

Multidimensional Arrays

Multidimensional Arrays

- The arrays, that we have seen so far, are all with one dimension.

```
int x[size];
```

**there is only
one subscript**

- A **multidimensional array** is an array with two or more dimensions.

**allocates memory
for size1 * size2 * ...
* sizeN data items**

```
double y[size1][size2];  
int z[size1][size2]...[sizeN];
```

**two dimensional
array**

**n-dimensional
array**

2-D Array: `int A[3][4]`

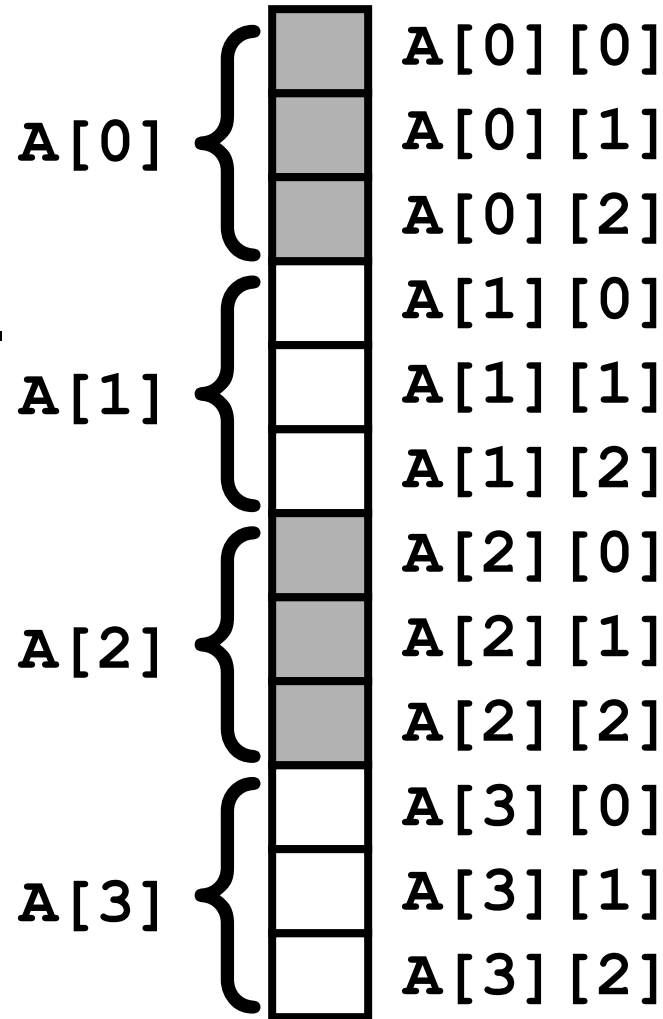
	Col 0	Col 1	Col 2	Col 3
Row 0	<code>A[0][0]</code>	<code>A[0][1]</code>	<code>A[0][2]</code>	<code>A[0][3]</code>
Row 1	<code>A[1][0]</code>	<code>A[1][1]</code>	<code>A[1][2]</code>	<code>A[1][3]</code>
Row 2	<code>A[2][0]</code>	<code>A[2][1]</code>	<code>A[2][2]</code>	<code>A[2][3]</code>

2-D Memory Organization

```
int A[4][3];
```

A is an array of size 4.

Each element of A is
an array of 3 chars



Multidimensional Arrays

- Two-dimensional arrays are used to represent tables of data, matrices, etc.
- Example:** Multiplication table:

```
#define NROWS 5
#define NCOLS 5
...
int table[NROWS][NCOLS];
int i, j;
for (i=0; i<NROWS, i++)
    for (j=0; j<NCOLS; j++)
        table[i][j]=i*j;
std::cout << table[2][3];
```

Columns

	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	6	8
3	0	3	6	9	12
4	0	4	8	12	16

Rows

Initialization of Multidimensional Arrays

- Multidimensional arrays can be initialized in their declarations, just like one-dimensional arrays are initialized. Instead of listing all table values in one list, these values are grouped by rows.
- Example:

```
char table[3][2]={ { 'A' , 'B' } ,  
                  { 'D' , 'G' } ,  
                  { '1' , '0' } } ;
```

Multidimensional Arrays as Function Arguments

- Multidimensional arrays can be passed to a function, as individual array elements or entirely, just like one-dimensional arrays.
- The size of the array, in one-dimensional case, is not specified in the function prototype
`void function1(int x[]);`
- However, with multidimensional arrays, the size of the array must be specified in the function prototype.
`void function2(int y[3][2][5]);`
- Therefore, only fixed-size multidimensional arrays can be passed into a function.
- Here, there is only one exception: the first size of the array can be omitted. Thus, the following function prototype is also valid:

```
void function2(int y[][2][5]);
```


Multidimensional Arrays as Function Arguments

- **Example:** Recall the multiplication table example. Let's also compute the sum of the numbers in each row of the table.

```
#include <stdio.h>
#define NROWS 5
#define NCOLS 5
void mult_table(int[NROWS][NCOLS]);
void sum_table(int[][NCOLS], int[]);

int main(){
    int table[NROWS][NCOLS];
    int sum_row[NROWS];
    mult_table(table);
    sum_table(table, sum_row);
    return 0;
}
```

Multidimensional Arrays as Function Arguments

 could be written as []

```
void mult_table(int matrix[NROWS][NCOLS]){
    /* compute the multiplication table */
    int i, j;
    for (i=0; i<NROWS; i++)
        for (j=0; j<NCOLS; j++)
            matrix[i][j]=i*j;
}
```

```
void sum_table(int matrix[][NCOLS], int arr[]){
    /* compute the sum of the numbers in each row */
    int i, j;
    for (i=0; i<NROWS; i++){
        arr[i] = 0;
        for (j=0; j<NCOLS; j++)
            arr[i] += matrix[i][j];
    }
}
```

Array Sorting and Binary Search


Sorting of Arrays

- Simple Algorithm for sorting arrays:
Bubble-Sort
 - The basic idea is to repeatedly compare neighboring objects and to swap them if they are in the wrong order.
 - Elements move upwards to their final positions like air-bubbles in water

Bubble-Sort in C++

- Given an array **a** of numbers with length **n** (the array consists of **n** elements)

```
for (int i=0; i<n-1; i++) {  
    for (int j=0; j<n-1-i; j++)  
        if (a[j+1] < a[j]) {  
            int tmp = a[j];  
            a[j] = a[j+1];  
            a[j+1] = tmp;  
        } /* if */  
    } /* outer for */
```



swap **a[j]** and **a[j+1]**

Bubble-Sort / Description

- The index j in the inner loop travels up the array, comparing adjacent entries in the array (at j and $j+1$), while the outer loop causes the inner loop to make repeated passes through the array.
- After the first pass, the largest element is guaranteed to be at the end of the array, after the second pass, the second largest element is in position, and so on.

Binary Search in Arrays

- A fast way to search a **sorted array** is to use a *binary search*.
- The idea is to look at the element in the middle:
 - If the key is equal to that, the search is finished.
 - If the key is smaller than the middle element, do a binary search on the first half.
 - If it's greater, do a binary search on the second half.
- (We denote the element we are searching for the “key”)

Binary-Search in C++

```
int binarySearch(int a[], int first, int last, int key) {  
    while (first <= last) {  
        int mid = (first + last) / 2; // compute mid point.  
        if (key > a[mid])  
            first = mid + 1; // repeat search in top half.  
        else if (key < a[mid])  
            last = mid - 1; // repeat search in bottom half.  
        else  
            return mid; // found it. return position  
    }  
    return -1; // failed to find key  
}
```

- **Parameters:**

- **a** : array of sorted (ascending) values.
- **first, last** : lower and upper subscript bounds
- **key** : value to search for.