

More on Strings in C++ Arrays

CS 16: Solving Problems with Computers I
Lecture #11

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- Heads-Up: Midterm #2 is NEXT Tuesday (11/14)
 - Covers everything from Lecture #7 thru Lecture #12
 - Of course, it's assumed you can work with if/else, loops, functions, etc...!

MIDTERM #2 IS COMING!

NOVEMBER 14_{th}!

- Material: Post-Midterm #1 *Lecture 7 thru 12*
 - Homework, Labs, Lectures, Textbook
- **Tuesday, 11/14** in this classroom
- Starts at **2:00pm **SHARP**** (come early)
- Ends at **3:15pm **SHARP****
- **BRING YOUR STUDENT IDs WITH YOU!!!**
- Closed book: no calculators, no phones, no computers
- Only 1 sheet (single-sided) of written notes
 - Must be no bigger than 8.5" x 11"
 - **You have to turn it in with the exam**
- **You will write your answers on the exam sheet itself.**



What's on Midterm #2?

- Test and Debugging Functions
- General Debug Techniques
- Numerical (all combos of bin, oct, hex, and dec) Conversions
- C-Strings
- Character Manipulators
- C++ Strings
 - ... and **all** its built-in member functions that we discussed
- File I/O
 - ... and **all** the many built-in member functions & manipulators that we discussed, both character-based and string/line-based
- Arrays (all that we'll cover this week in class*)

From the book:
Chapter 5.4, 5.5, 6, 7*, and 8.1, 8.2

Lecture Outline

- **Strings (Ch. 8.1, 8.2)**
- Built-in string functions
- **Arrays (Ch. 7)**

Built-In String Member Functions

- Search functions
 - **find, rfind, find_first_of, find_first_not_of**
- Descriptor functions
 - **length, size**
- Content changers
 - **substr, replace, append, insert, erase**

Descriptor Functions: **length** and **size**

- The **length** function returns the length of the string
- The member function **size** is the same exact thing...
 - So, if **string str1 = “Mama Mia!”**,
then **str1.length() = 9**
and **str1.size() = 9** also

Example – what will this code do?:

```
string name = “Bubba Smith”;  
for (int i = name.length(); i > 0; i--)  
    cout << name[i-1];
```

Content Changers: **append**

- Use function **append** to append one string to another

```
string name1 = " Max";  
string name2 = " Powers";  
cout << name1.append(name2); // Displays " Max Powers"
```
- Does the same thing as: **name1 + name2**

Content Changers: **erase**

- Use function **erase** to clear a string to an empty string
- One use is:
name1.erase() -- Does the same thing as: **name1 = ""**
- Another use is:
name1.erase(*start position, how many chars to erase*)
 - Erases only part of the string
 - Example:

```
string s = "Hello!";  
cout << s.erase(2, 2);
```

// Displays "Heo!"

Content Changers: **replace** and **insert**

- Use function **replace** to replace part of a string with another
 - Popular Usage:
`string.replace(start position, # of places after start position to replace, replacement string)`
- Use function **insert** to insert a substring into a string
 - Popular Usage:
`string.insert(start position, insertion string)`

Example:

```
string country = "Back in the USSR"; // length is 16
cout << country.replace(14, 2, "A");    // Displays "Back in the USA"
cout << country.insert(15, "BC");      // Displays "Back in the USABC"
```

Content Changers: **substr**

- Use function **substr** (short for “substring”) to extract and return a substring of the **string** object

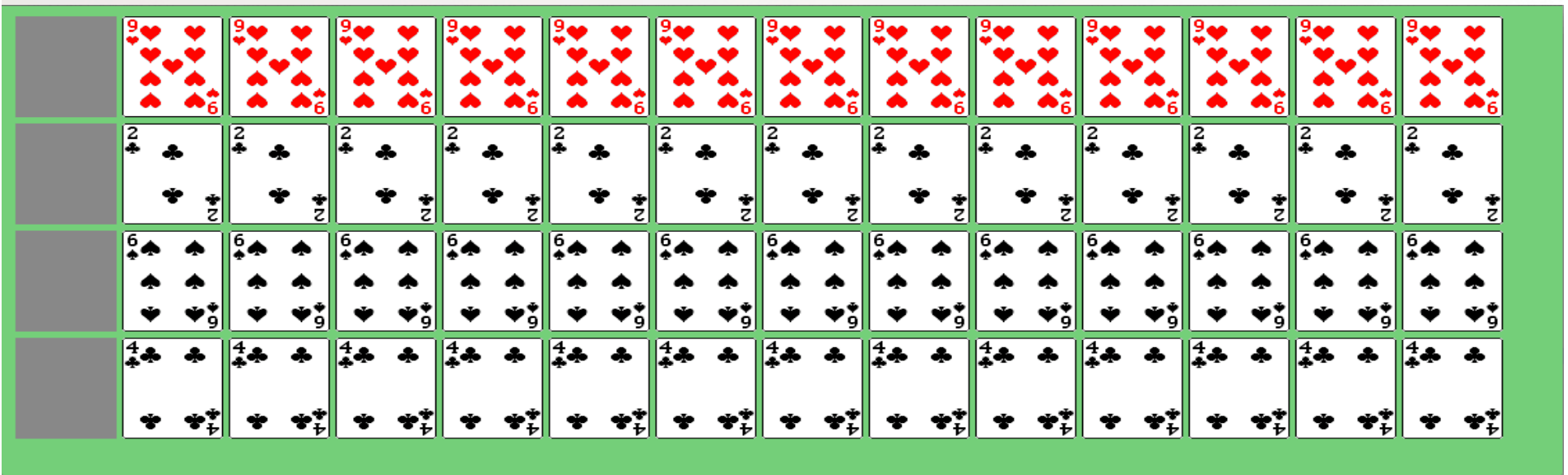
– Popular Usage:

`string.substr(start position, # of places after start position)`

Example:

```
string city = "Santa Barbara";  
cout << city.substr(3, 5)    // Displays "ta Ba"
```


ARRAYS



Introduction to Arrays

- An array is used to process a collection of data of the same type
 - Examples: A list of people's last names
 A list of numerical measurements
- Why do we need arrays?
 - Imagine keeping track of 1000 test scores in memory!
 - How would you name all the variables?
 - How would you process each of the variables?

Declaring an Array

- An array, named **score**, containing five variables of type **int** can be declared as

```
int score[5];
```

- This is like declaring 5 variables of type int:

```
int score[0], score[1], ... , score[4]
```
- The value in [brackets] is called: ***a subscript*** or ***an index***
- Note the **size** of the array is the **highest index value + 1**
 - Because indexing in C++ starts at 0, not 1
 - The index can be an integer data type variable also (e.g. score[n])

Array Variable Types

- An array can have indexed variables of ***any type*** – they just all have to be the **SAME** type
- Use an indexed variable the same way an “ordinary” variable of the base type would be
 - If the array is type int, for example, you can do
score[4] + score[5] * score[6], etc...

Indexed Variable Assignment

- To assign a value to an indexed variable, use the assignment operator (just like with other variables):

```
int n = 2;  
score[n + 1] = 99;
```

In this example, variable `score[3]` is assigned the value 99

Loops And Arrays


- for-loops are commonly used to step through arrays

Example:

First index is 0

Last index is (size – 1)

```
int max = 9;
for (i = 0; i < 5; i++)
    cout << score[i] << " off by "
         << (max - score[i]) << endl;
```



could display the difference between each score and the maximum score stored in an array

Declaring An Array

- When you declare an array, you **MUST** declare its **size** as well!

```
int MyArray[5];  
//Array declared has 5 non-initialized elements
```

{ ... } used for full-array initializations



```
int MyArray[] = {1, 2, 5, 7, 0};  
// Array declared has 5 initialized elements
```

```
int MyArray[5] = {1, 2, 5, 7, 0};  
// This is ok too!
```

Initializing Arrays

- To initialize an array when it is declared
 - The values for the indexed variables are enclosed in braces and separated by commas

- Example: `int children[3] = {2, 12, 1};`

Is equivalent to:

```
int children[3];  
children[0] = 2;  
children[1] = 12;  
children[2] = 1;
```


Constants and Arrays

- You can use **constants** (but not variables) to **declare** size of an array
 - Allows your code to be easily altered for use on a smaller or larger set of data

Example:

```
const int  NUMBER_OF_STUDENTS = 50; // can change this later
int  score[NUMBER_OF_STUDENTS];

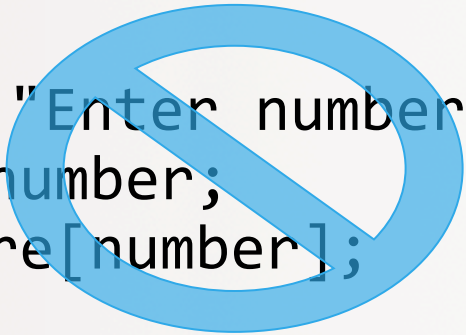
...
for ( int i = 0; i < NUMBER_OF_STUDENTS; i++)
    cout << score[i] << endl;
```

- To make this code work for any number of students, simply change the value of the constant in the 1st line...

Variables and Declarations

- Most compilers **do not allow** the use of a **variable** to **declare** the size of an array

Example: `cout << "Enter number of students: ";`
 `cin >> number;`
 `int score[number];`

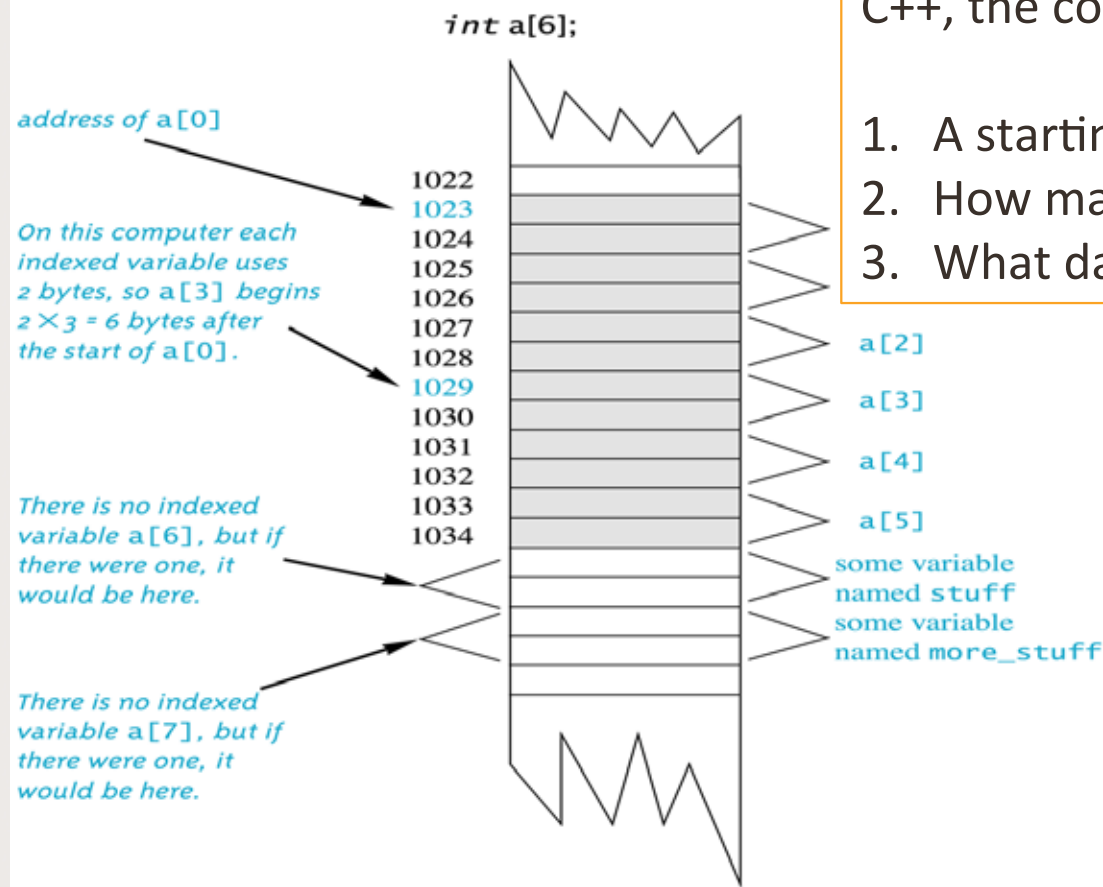


- This code is illegal on many C++ compilers
- Later we will take a look at **dynamic arrays** which do support this concept (but using *pointers*)

Arrays and Computer Memory

- When you declare the array `int a[6]`, the compiler...
 - Reserves memory for six variables of type `int` starting at some memory address (that the compiler picks)
 - The variables are stored one after another (adjacent in memory)
 - The address of `a[0]` is remembered
 - The addresses of the other indexed variables is not remembered (b/c there's no need to!)
- If the compiler needs to determine the address of `a[3]`
 - It starts at `a[0]` (*it knows this address!*)
 - It counts past enough memory for three integers to find `a[3]`

An Array in Memory



When reserving memory space for an array in C++, the compiler needs to know **just 3 things**:

1. A starting address (location)
2. How many elements in array
3. What data type the array elements are

Array Index Out of Range

- A common error is using a nonexistent index
- Index values for `int a[6]` are the **values 0 through 5**
- An index value that's not allowed by the array declaration is called ***out of range***
- Using an out of range index value **does not** produce an error message by the compiler!!!
 - It produces a WARNING, but the program will often (but NOT always) give a run-time error
 - So, DON'T rely on the compiler catching your mistakes! **Be Proactive!**

Out of Range Problems

- If an array is declared as: `int a[6];`
and an integer is declared as: `int i = 7;`
- Executing the statement: `a[i] = 238;` causes...
 - The computer to calculate the address of the illegal `a[7]`
 - This address could be where some **other** variable in the program is stored
 - The value 238 will be stored at the address calculated for `a[7]`
 - You could get run-time errors OR YOU MIGHT NOT!!!
- *This is bad practice! Keep track of your arrays!*

Default Values

- If too few values are listed in an initialization statement
 - The listed values are used to **initialize the first** of the indexed variables
 - The remaining indexed variables are initialized to a **zero** of the base type
- Example: `int a[10] = {5, 5};` // Note array size is given
initializes `a[0]` and `a[1]` to **5**
and `a[2]` through `a[9]` to **0**

NOTE:

This is called an ***extended initializer list*** and it only works in the latest versions of C++ compilers. So make sure you compile with the **`-std=c++11`** option when using **`g++`**.

Is this OK?

```
int num[ ] = {0, 0, 0};
```

- When an array is initialized, C++ allows you to leave the square brackets empty!!!
 - But... but... but... don't I have to *explicitly* define the size?!?!?!?!?!?!?!?!?!?!?!?!?!?!?!?
- In this case, the compiler will **automatically** assume a size for the array that matches the number of values included between the braces { }
 - So, you're *still* explicitly defining the size!
- So, the example makes the array **num** size 3 automatically.
- This shortcut is mostly used for initializing small arrays
 - Impractical, otherwise...

Range-Based For Loops

- C++11 includes a new type of **for loop**:
The range-based for-loop simplifies iteration over every element in an array. The syntax is shown below:

```
for (datatype varname : array)
{
    // varname is successively set to each
    // element in the array
}
```

Range-Based For Loop Example

- The following code outputs: **2 4 6 8**

```
int arr[ ] = {2, 4, 6, 8};  
for (int x : arr)  
{  
    cout << x;  
    cout << " ";  
}
```

Arrays in Functions

- Indexed variables can be arguments to functions
- Example: If a program contains these declarations:

```
void my_function(int x);  
int i, n, a[10];
```

Variables `a[0]` through `a[9]` are of type **int**, so making these calls **IS** legal:

```
my_function( a[0] );  
my_function( a[3] );  
my_function( a[i] );
```

BUT! This call is **NOT** legal:

```
my_function( a[] );    or    my_function( a );
```

Arrays as Function Arguments

- You *can* make an entire array a formal parameter for a function
 - That is, an input to a function
- But you *cannot* make an entire array the RETURNED value for a function
 - That is, an output from a function
- An array parameter
 - is not a call-by-value parameter or a call-by-reference parameter
 - Although it **behaves** much like call-by-reference parameters

Passing an Array into a Function

- An array parameter is indicated using **empty brackets** in the parameter list such as

```
void fill_up(int a[], int size);
```

Function Calls With Arrays

- If function **fill_up** is *declared* in this way (uses [] !!!)
`void fill_up(int a[], int size);`
- and array **score** is declared this way:
`int score[5], number_of_scores = 5;`
- **fill_up** is *called* in this way (note: no [] !!!)
`fill_up(score, number_of_scores);`
- Note that the array values can be *changed* by the function
 - Even though it “looks like” it’s being passed-by-value

Function with an Array Parameter

Function Declaration

```
void fill_up(int a[], int size);  
//Precondition: size is the declared size of the array a.  
//The user will type in size integers.  
//Postcondition: The array a is filled with size integers  
//from the keyboard.
```

Function Definition

```
//Uses iostream:  
void fill_up(int a[], int size)  
{  
    using namespace std;  
    cout << "Enter " << size << " numbers:\n";  
    for (int i = 0; i < size; i++)  
        cin >> a[i];  
    size--;  
    cout << "The last array index used is " << size << endl;  
}
```

Array Argument Details

- Recall: What does the computer know about an array?
 - The base **type**
 - The **address of the first** indexed variable
 - The **number** of indexed variables
- What does a function need know about an array argument?
 - The base **type**
 - The **address of the first** indexed variable

Array Parameter Considerations

- Because a function **does not know the size** of an array argument...
 - The programmer should include a formal parameter that specifies the size of the array
 - The function can process arrays of various sizes
 - Example: function **fill_up** from on pg. 392 of the textbook can be used to fill an array of any size:

```
fill_up(score, 5);  
fill_up(time, 10);
```

But...

IS there a way to CALCULATE the Size of an Array?

- Yes, there is...
- But, practically-speaking, why would you do that?
- More on that later...
- For now, get used to the idea of passing the size of an array into a function that has the array as argument.

const Modifier

- Array parameters allow a function to change the values stored in the array argument
 - Similar to how a parameter being passed by reference would be
 - This is because an array is structured by C++ *as a pointer* (more on this later)
- If you want a function to ***not change*** the values of the array argument, use the modifier **const**
- An array parameter modified with **const** is called a *constant array parameter*
 - Example:

```
void show_the_world(const int a[ ], int size);
```

Using **const** With Arrays

- If **const** is used to modify an array parameter:
 - **const** has to be used in *both* the function *declaration* and *definition*
 - The compiler will issue an error if you write code that changes the values stored in the array parameter

Returning An Array

- Recall that functions can return a value of type int, double, char, ..., or even a class type (like string)
- **BUT functions cannot return arrays**
- We'll learn later how to return a ***pointer*** to an array instead...

YOUR TO-DOs

- ☐ HW 6 due Thu. 11/9
- ☐ Lab 6 due Fri. 11/10
- ☐ Visit Prof's and TAs' office hours if you need help!
- ☐ Recycle!

</LECTURE>