Character Functions & Manipulators
Arrays in C++

CS 16: Solving Problems with Computers I
Lecture #10

Ziad Matni
Dept. of Computer Science, UCSB

Lecture Outline

- Useful character manipulators & functions
- Arrays in C++

Member Functions: **get** and **getline**

See demo files:
get_example.cpp
getline_example.cpp

- Allow you to use input streams that include white-spaces
  - ***Unlike cin***, which separates inputs by white-spaces
  - Recall: white-space = space, tab, newline characters

.get

```
char c_fin, c_cin;
ifstream inf;

inf.get(c_fin);
cin.get(c_cin);
```

.getline

```
string fstring, cin_string;
ifstream inf;

getline(inf, fstring);
getline(cin, cin_string);
```

getline function

- For standard inputs, **cin** is fine: but it ignores space, tabs, and newlines
- Sometimes, you want to get the ***entire line of data!***
  - ***And stop at the newline***
- Best to use the function **getline** for that purpose.
- You have to include the **<iostream>** library (which you likely already do!)
- Popular Usage:

```
getline(ifstream_object, string);
getline(cin, string);
```

Additional Note About **getline**

- You can customize what character a getline stops "getting" info
  - You can define the "character delimiter"
  - By default, that's a newline char

Example:

```
getline(cin, VariableX, 'm')    //stops at the char 'm'
```

If the standard input is "Hello, I must be going",
then **VariableX** will be "Hello, I "

Character Functions

- Several predefined functions exist to facilitate working with characters
- The **cctype** library is required for most of them

```
#include <cctype>
using namespace std;
```

The **toupper** Function

- **toupper** returns the argument's upper case character
  - `toupper('a')` returns 'A'
  - `toupper('A')` returns 'A'

DOES NOT WORK WITH STRINGS!

IT'S FOR CHARACTERS ONLY!

The **tolower** Function

- Similar to **toupper** function…
- **tolower** returns the argument's lower case character
  - `tolower('a')` returns 'a'
  - `tolower('A')` returns 'a'

The **isspace** Function

- **isspace** returns *true* if the argument is a whitespace

  (spaces, tabs, and newlines)

  – So, **isspace(' ')** returns true, so does **isspace('\n')**

  Example:

```
if (isspace(next) )
   cout << '-';
else
   cout << next;
```

Prints a '-' if next contains a space, tab, or newline character

Some Predefined Character Functions in cctype (*part 2 of 2*)

Function	Description	Example

`isupper(`*Char_Exp*`)`	Returns *true* provided *Char_Exp* is an uppercase letter; otherwise, returns *false*.	`if (isupper(c))`
		`  cout << c << " is uppercase.";`
		`else`
		`  cout << c`
		`  << " is not uppercase.";`

`islower(`*Char_Exp*`)`	Returns *true* provided *Char_Exp* is a lowercase letter; otherwise, returns *false*.	`char c = 'a';`
		`if (islower(c))`
		`  cout << c << " is lowercase.";`
		Outputs: a is lowercase.

`isalpha(`*Char_Exp*`)`	Returns *true* provided *Char_Exp* is a letter of the alphabet; otherwise, returns *false*.	`char c = '$';`
		`if (isalpha(c))`
		`  cout << c << " is a letter.";`
		`else`
		`  cout << c`
		`  << " is not a letter.";`
		Outputs: $ is not a letter.

`isdigit(`*Char_Exp*`)`	Returns *true* provided *Char_Exp* is one of the digits '0' through '9'; otherwise, returns *false*.	`if (isdigit('3'))`
		`  cout << "It's a digit.";`
		`else`
		`  cout << "It's not a digit.";`
		Outputs: It's a digit.

`isspace(`*Char_Exp*`)`	Returns *true* provided *Char_Exp* is a whitespace character, such as the blank or newline symbol; otherwise, returns *false*.	`//Skips over one "word" and`
		`//sets c equal to the first`
		`//whitespace character after`
		`//the "word":`
		`do`
		`{`
		`  cin.get(c);`
		`} while (! isspace(c));`

Character Manipulators Work Too!

- Include **<cctype>** to use with, for example, **toupper()**

```
string s = "hello";
s[0] = toupper(s[0]);
cout << s;   // Will display "Hello"
```

- …or to use with **tolower()**

```
string s = "HELLO";
for (int i=0; i < 5; i++) s[i] = tolower(s[i]);
cout << s;   // Will display "hello"
```

Manipulators

- A type of function called in a nontraditional way
- Manipulators, in turn, *call member functions*
  - May or may not have arguments to them
- Used after the insertion operator (**<<**) as if the manipulator function call is an output item

The **setw** Manipulator

- setw sets spaces for output: only effective for one use
  - Found in the library **<iomanip>**

- Example:

```
cout << "Start" << setw(4) << 10
                << setw(4) << 20 << setw(6) << 30;
```

produces: Start 10 20 30

- *The 1st setw(4) ensures 4 spaces between "Start" and 10, INCLUSIVE of the spaces taken up by 10.*
- *The 2nd setw(4) ensures 4 spaces between 10 and 20, INCLUSIVE of the spaces taken up by 20.*
- *The 3rd setw(6) ensures 6 spaces between 20 and 30, INCLUSIVE of the space taken up by 30.*

Converting Data Types in C++
`stoi` `to_string`

`stoi()` String-to-Integer conversion

- Found in <string> library.
- Takes string as argument and returns int type.
- Example: `int x = stoi("66")  // x = 66`
- If the string is NOT a number representation, **it will cause a runtime error!**

`to_string()` Number-to-String conversion

- Found in <string> library.
- Takes int or double as argument and returns string type.
- Example: `string y = to_string(6.32)  // y = "6.32"`

Converting Data Types in C++
Combining Characters with Strings

- Consider this code using C++ Strings:

```
string msg1 = "Hello", msg2 = "World";
char sp = ' ';   // space character
string msg3 = msg1 + msg2;
string msg4 = msg1 + sp + msg2;
string msg4 = msg1 + sp + msg2[0];
string msg4 = msg1[0] + sp + msg2[0];
```

Compiles!

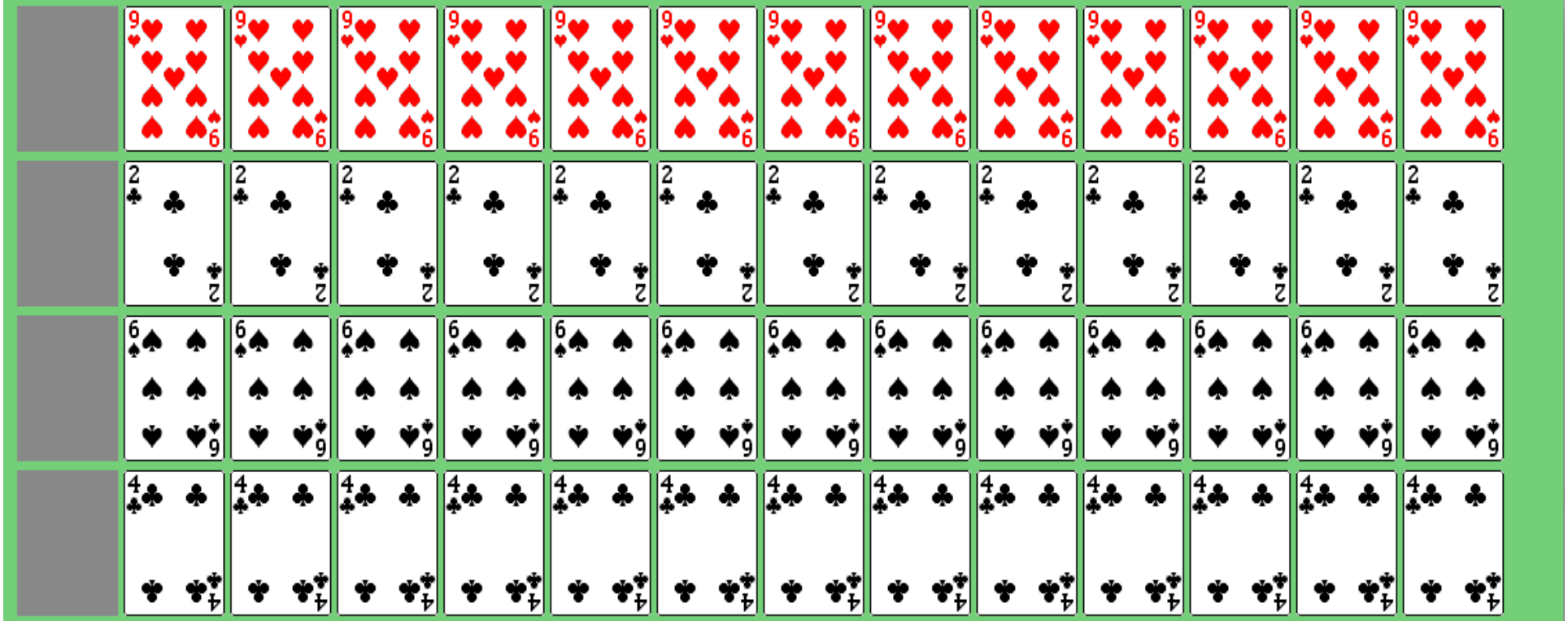
Also Compiles!

Also Compiles!

Does NOT Compile!

- You can create a string that is a concatenation of strings + characters
- You CANNOT create a string out of ***only*** characters!
  - Concatenation is just for strings - not for characters.

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

```
int score[5];
// Declares an array of ints called score that has 5 elements:
// score[0], score[1], score[2], score[3], score[4]
```

- 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

Loops And Arrays

- for-loops are commonly used to step through arrays

Example:

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

Last index is (size – 1)

First index is 0

displays 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!

{ … } used for full-array initializations

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

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

- It's recommended 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;
```

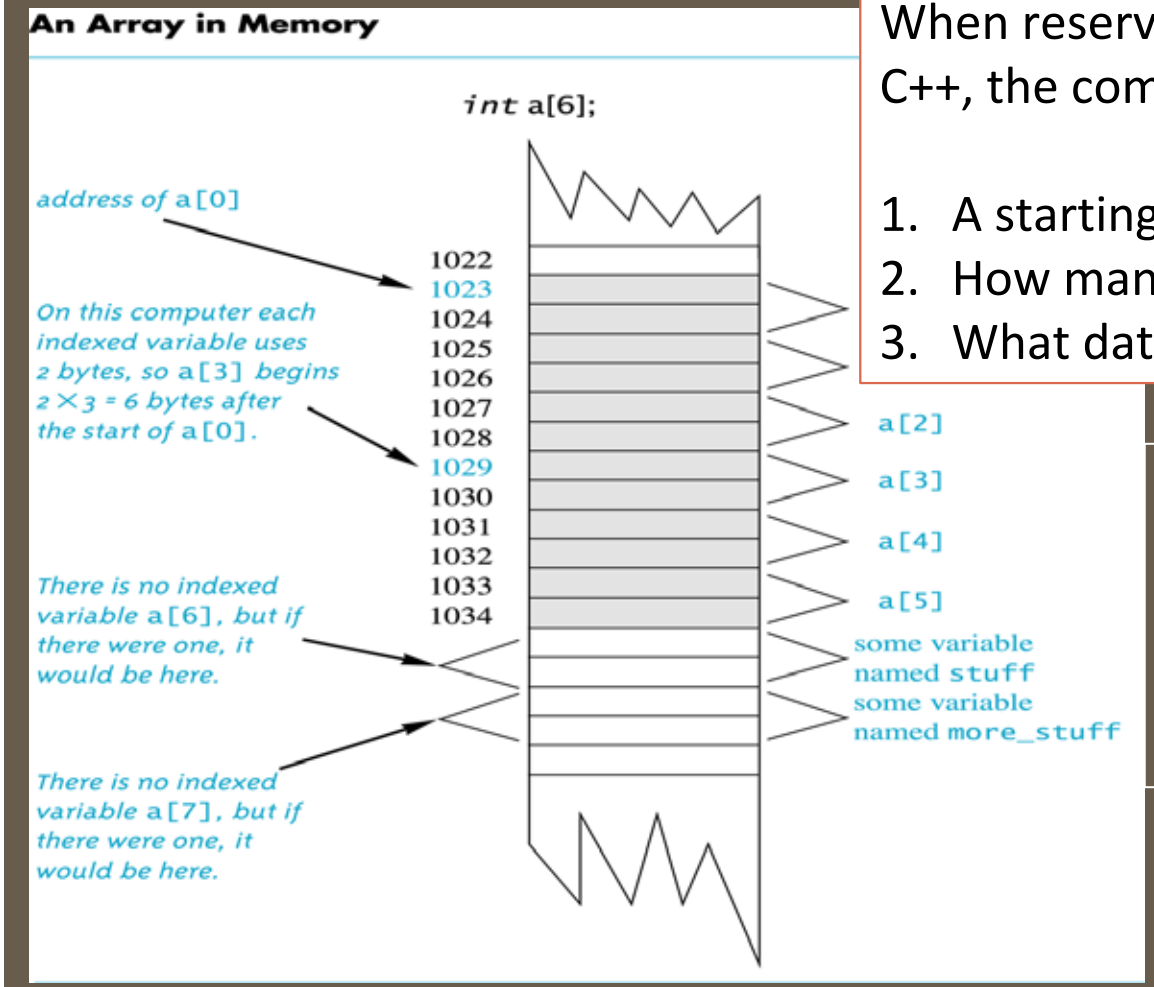
Constants and Arrays

- You can use **variables** as indices in arrays, ***BUT NOT*** to declare them!
- However, you can use **constants** to ***declare*** size of an array

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…



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

If the compiler needs to determine the address of **a[3]**, for example

- It starts at **a[0]** *(it knows this address!)*
- It counts past enough memory for three integers to find **a[3]**

Array Index Out of Range

- **A common error by programmers 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** always produce an error message by the compiler!!!
  - It produces a WARNING, but the program will often give a run-time error
  - So, DON'T rely on the compiler catching your mistakes! **Be Proactive!**

Out of Range Problems

See demo files:
basic_arrays.cpp

- Let's say we have the following: `int a[6], i = 7;`
- Then we execute the statement: `a[i] = 238;`
- This 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]

- **Congrats! You've now messed with the integrity of computer memory!**
- You could get run-time errors OR YOU MIGHT NOT!!! (unpredictable)
- *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 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 (version 11 or later).

Range-Based For Loops

- C++11 (and later) includes a new type of for loop:
  The range-based for-loop simplifies iteration over every element in an array.

- For example, the following code outputs: **2 4 6 8**

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

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, as an **input** to a function
- But you ***cannot*** make an entire array the RETURNED value for a function
  - That is, as an **output** from a function
- An array parameter ***behaves*** much like a call-by-reference parameter

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 (note: 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 – it's actually being passed-by-reference. We'll discuss this more with "pointers" another time…

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 not with regular arrays
- You will want to use “dynamic arrays”
  - We’ll talk about those later on with “pointers”

- 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

See demo files:
array_function.cpp

- Array parameters allow
  a function to change the values stored in the array arg.
  - Similar to how a parameter being passed by reference would be
- If you want a function to ***not change*** the values of the array argument, use the modifier **const**
- An array param. modified w/ **const** is called a *constant array parameter*
  - Example:

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

- If **const** is used to modify an array parameter:
  it has to be used in **both** the function declaration and definition

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

- ❑ Begin Lab6 on Wednesday
- ❑ Do HW10 by next Thursday

- ❑ Visit Prof's and TAs' office hours if you need help!

- ❑ Practice all your skills

</LECTURE>