Arrays in C++

CS 16: Solving Problems with Computers I
Lecture #7 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 1 of 3

Lecture Outline

- Additional info on
  - Command Line Arguments

- Arrays in C++
  - Declaring, initializing
  - Use in functions
  - Multi-dimensional arrays

Example

Remember:
argv[0] holds the program name (not so useful).
argv[1] has the actual first argument from your command line.
argv[2] has the 2nd argument, etc… etc…
argv[] are of **char*[]** type and need to be converted before used as numbers

argc is the number of arguments after the program name
(it's automatically detected)

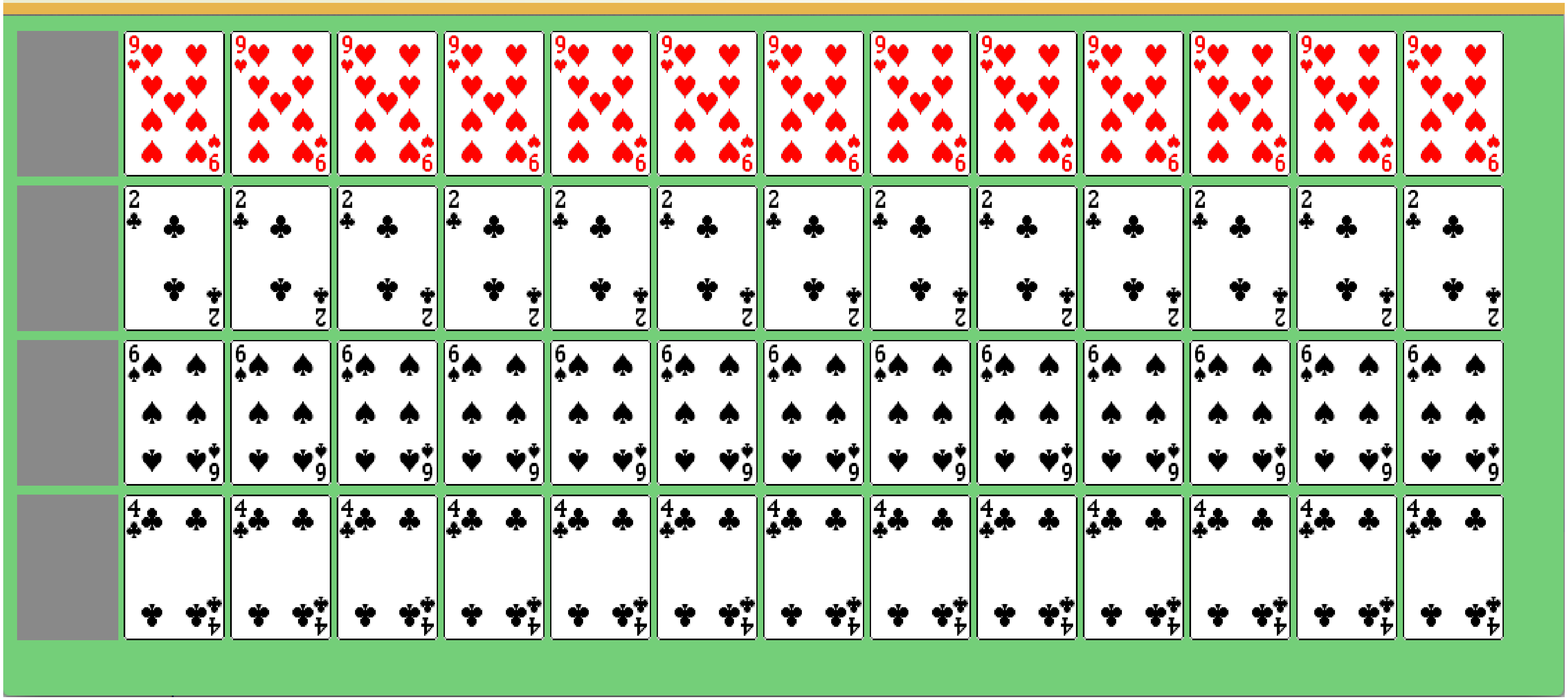
```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        cerr << "Error! You entered the wrong number of arguments!\n";
        exit(1);
    }
    int num1 = atoi(argv[1]);   //atoi() converts the argv[] element into an int
    int num2 = atoi(argv[2]);
    int add = num1 + num2;
    int prod = num1 * num2;
    cout << "The sum is: " << add << "\nThe product is: " << prod << endl;

    return 0;
}
```

Watch Out For…

- The use of **cerr** vs **cout** (esp. in current lab)
  - Use **cerr** when relaying **error messages**
  - Use **cout** for **regular standard output**

- When you create your programs, test them with as many different scenarios and "edge cases" as you can
  - So that you can catch errors and understand where/why they occur

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]
```

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

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!

```
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!
```

{ … } used for full-array initializations

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;
```

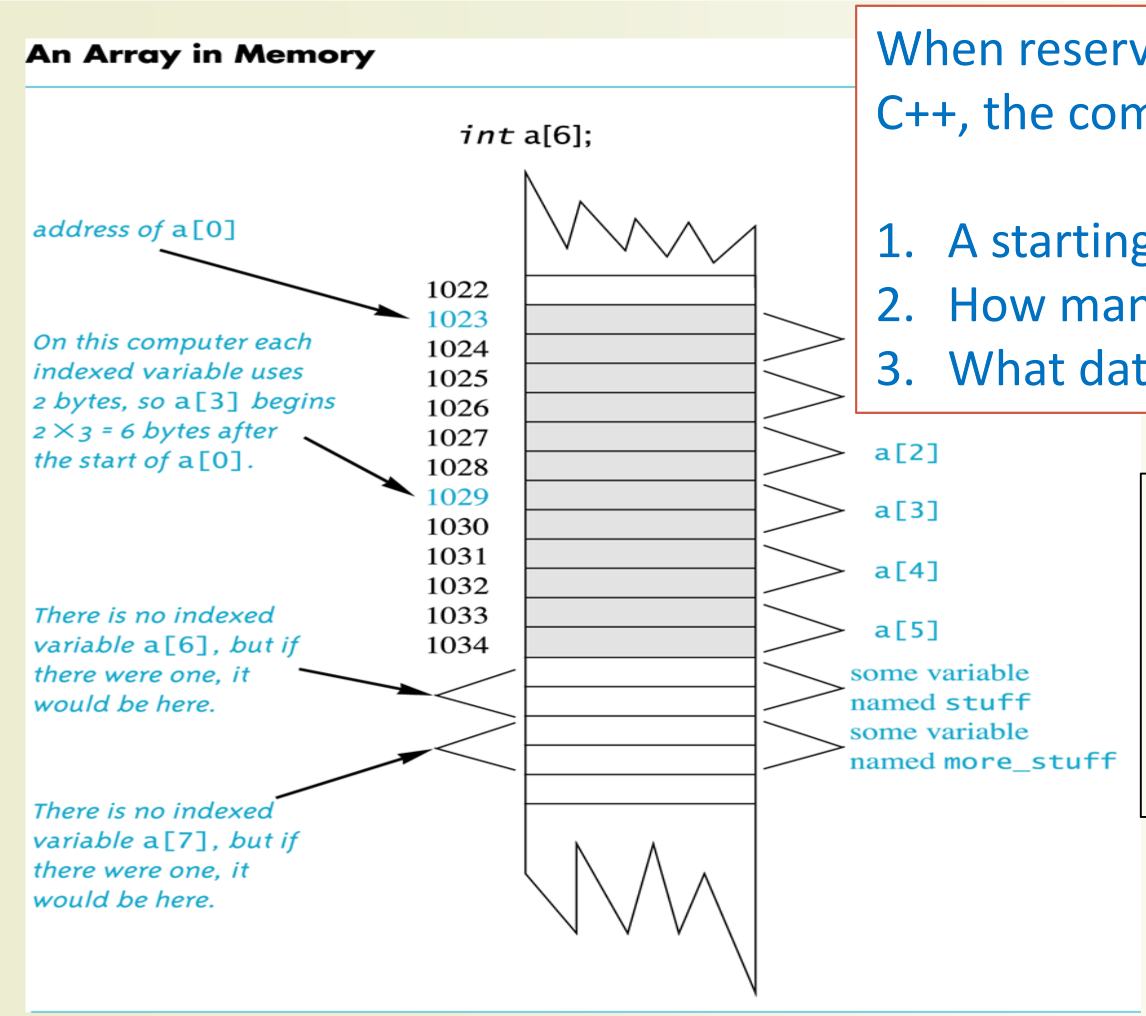
Constants and Arrays

- 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,
  you'd have to change the value of the constant in the 1st line each time
  - You cannot do this using, for example, **cin**… ☹☹☹



An Array in Memory

int a[6];
address of a[0]
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
On this computer each indexed variable uses 2 bytes, so a[3] begins 2 × 3 = 6 bytes after the start of a[0].
There is no indexed variable a[6], but if there were one, it would be here.
There is no indexed variable a[7], but if there were one, it would be here.
a[2]
a[3]
a[4]
a[5]
some variable named stuff
some variable named more_stuff

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 ***another*** variable in the program is stored! (which you might need!)
  - The value 238 ***will be stored*** at the address calculated for a[7], erasing what was on there
- **Congrats! You've now messed with the integrity of computer memory!**
- You could get run-time errors OR YOU MIGHT NOT!!! (it's totally unpredictable)
- *This is bad practice! Keep track of your arrays! (C++ is infamous for this…)*

END OF PART 1

Arrays in C++

CS 16: Solving Problems with Computers I
Lecture #7 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 2 of 3

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 will give this output: **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]; // this line is inside main()
```

Then, 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] );         // This is ok
```

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]; // this line is inside main()
```

BUT! These kinds of function calls are **NOT** legal:

```
my_function( a[] );   // Not ok because a[] is not an int
my_function( a );     // Not ok because a is not an int
```

Arrays as Function Arguments

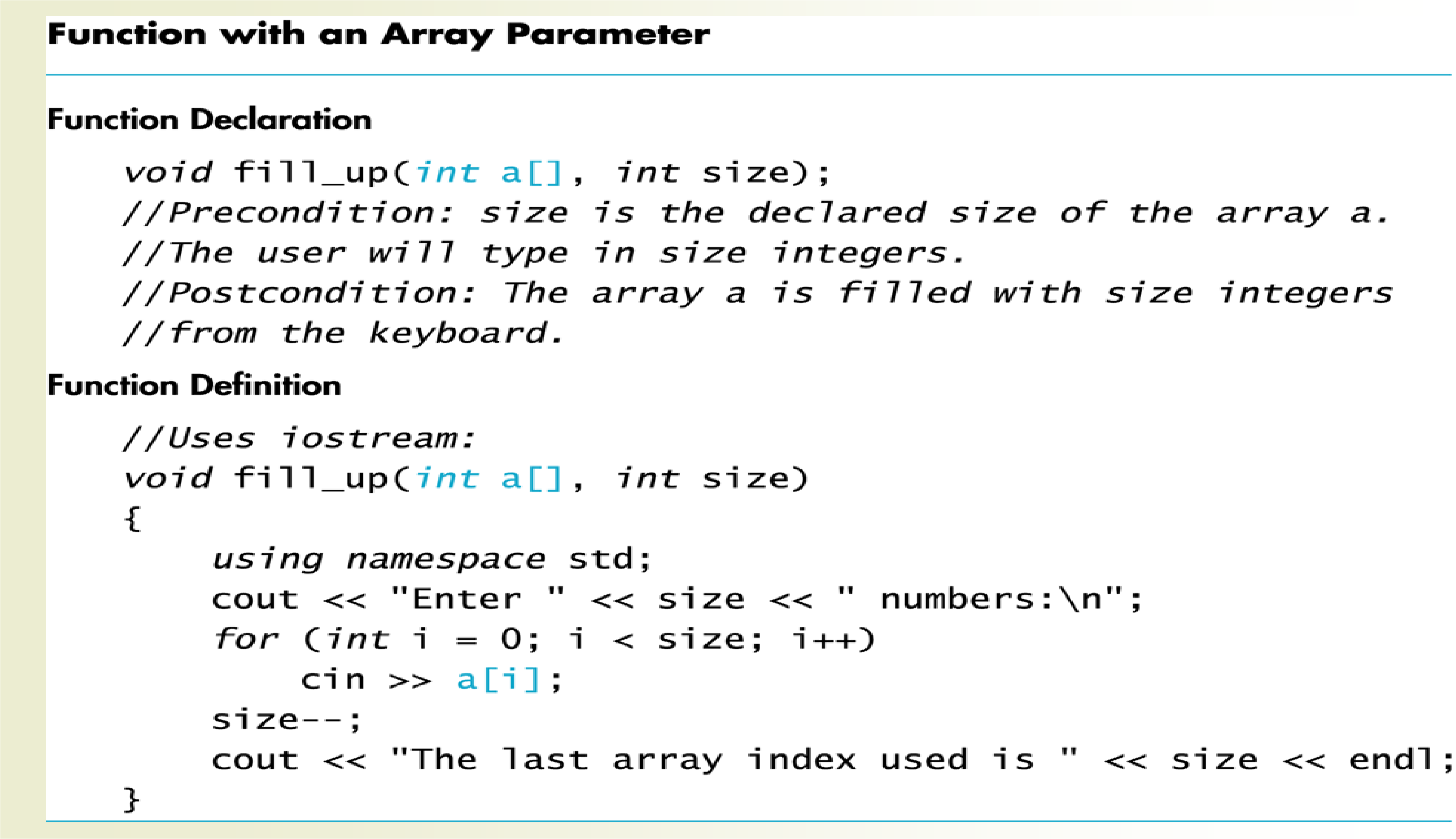
- You ***can*** make an entire array an argument in a function
  - i.e., as an **input** to the function
- But you ***cannot*** make an entire array be the **RETURNED** value for a function
  - i.e., as an **output** from a function
- An array parameter ***behaves*** much like a call-by-reference parameter

Passing an Array into a Function

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

```
    void fill_up(int arr[], int size);
// you have pass the array AND its size as fun. arguments
```

- Note that **`arr[]`** is the array
- While **arr** is an int variable that will contain the **MEMORY ADDRESS** of the start of the array (i.e. the memory address of **`arr[0]`**).



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 **compiler** know about an array?
  - The base **type** (e.g. int, double, etc…)
  - 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 a new type of variable: "*dynamic arrays*"
  - Covered in CS 24
- 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 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

See demo files:
array_func.cpp

Function Calls With Arrays

- If function **fill_up** is *declared* in this way

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

(note: use [ ] in dec./def. header!)

- and array **score** is declared this way:

```
int score[5], number_of_scores = 5;
```

- **fill_up** is *called* in this way

```
fill_up(score, number_of_scores);
```

(note: do **not use** [ ] in fun. call!)

- 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 next week…

END OF PART 2

Arrays in C++

CS 16: Solving Problems with Computers I
Lecture #7 PRE-RECORDED

Ziad Matni
Dept. of Computer Science, UCSB

Part 3 of 3

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**
  - They can change them, but there is no "type" to return them, per se

- You have to return a POINTER to an array from a function
  - Pointers are covered in CS 24

Summary Difference

```
void thisFunction(int arr[ ], int size);
```

Array “arr” gets passed and whatever changes are done inside the function will result in changes to “arr” where it’s called.

```
void thisFunction(const int arr[ ], int size);
```

Array “arr” gets passed BUT whatever changes are done inside the function will NOT result in changes to “arr” where it’s called.

```
int* thisFunction(int arr[ ], int size);
```

Array “arr” gets passed and whatever changes are done inside the function will result in changes to “arr” where it’s called. ADDITIONALLY, a new *pointer* to an array “thisFunction” is passed back ***(You are NOT responsible to know this for CS 16!!)***

Programming With Arrays

- Arrays are a little inflexible in C++
- You need to declare the size of an array
  - The size cannot be a simple int variable

- Sometimes the size of an array is *not known* when the program is written
  - But we need the size to be defined… so what then?
  - Don’t use basic arrays…
    - Book has a “partially filled arrays” method that works, but it’s not efficient
  - Either **vectors** or **dynamic arrays** work efficiently for that

Multi-Dimensional Arrays

- C++ allows arrays with multiple index dimensions (have to be same type, tho…)
- EXAMPLE: `char page[30][100];`
  declares an array of characters named **page**
  - **page** has two index values:
    The 1st ranges from 0 to 29
    The 2nd ranges from 0 to 99
  - Each index in enclosed in its own brackets

[0][0]	[0][1]	…	[0][98]	[0][99]
[1][0]	[1][1]	…	[1][98]	[1][99]
…	…	…	…	…
[28][0]	[28][1]	…	[28][98]	[28][99]
[29][0]	[29][1]	…	[29][98]	[29][99]

- Page can be visualized as an array of 30 rows and 100 columns
  - **page** is actually an array of size 30
  - **page's base type** is an array of 100 characters

Program Example: Grading Program

- Grade records for a class can be stored in a two-dimensional array

- A class with 4 students and 3 quizzes the array could be declared as

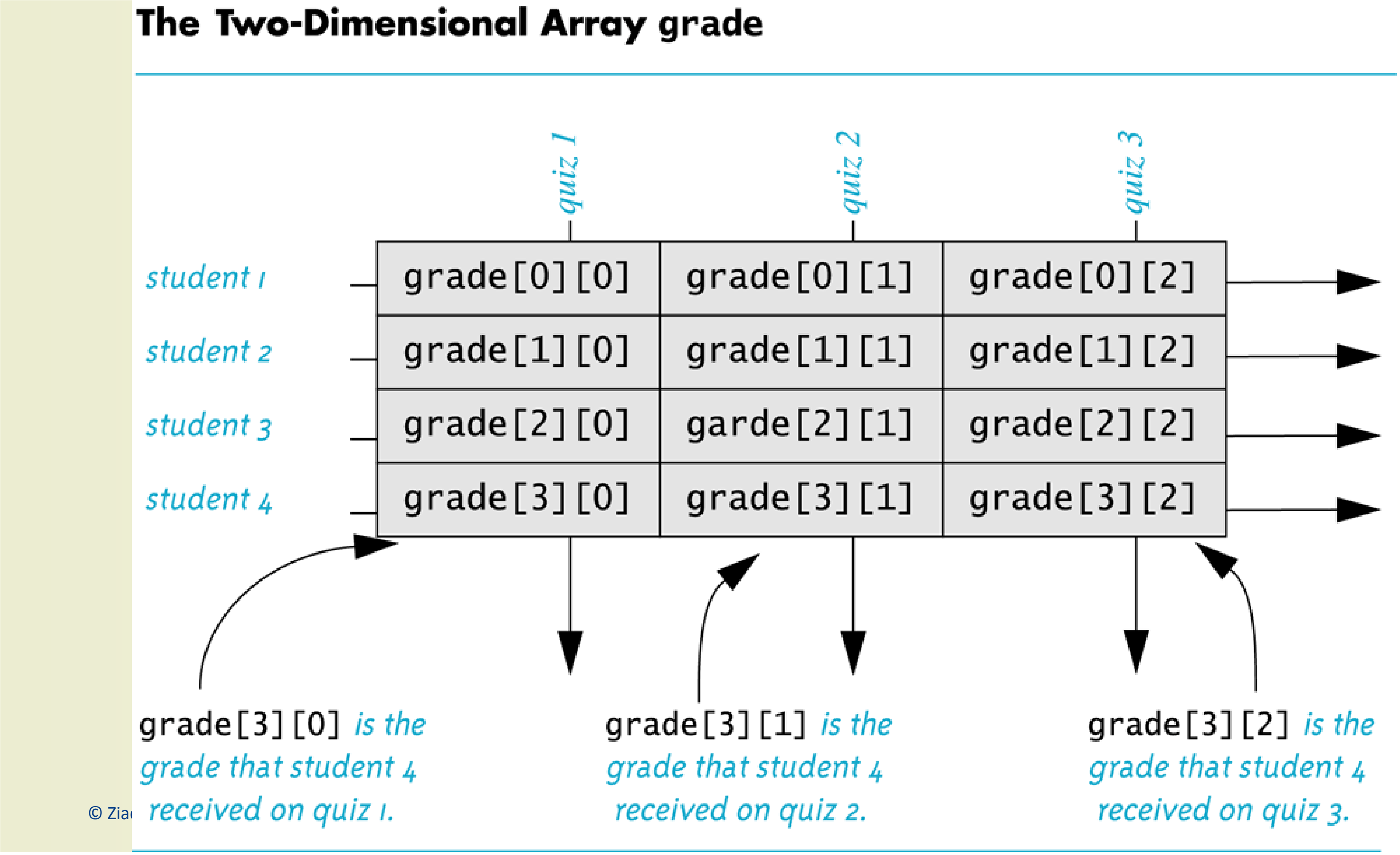
```
int grade[4][3];
```

Each student (0 thru 3) has 3 grades (0 thru 2)

  – The first array index refers to the number of a student

  – The second array index refers to a quiz number

- Your textbook, Ch. 7, Display 7.14 has an example



The Two-Dimensional Array grade

	quiz 1	quiz 2	quiz 3
student 1	`grade[0][0]`	`grade[0][1]`	`grade[0][2]`
student 2	`grade[1][0]`	`grade[1][1]`	`grade[1][2]`
student 3	`grade[2][0]`	`garde[2][1]`	`grade[2][2]`
student 4	`grade[3][0]`	`grade[3][1]`	`grade[3][2]`

`grade[3][0]` *is the grade that student 4 received on quiz 1.*

`grade[3][1]` *is the grade that student 4 received on quiz 2.*

`grade[3][2]` *is the grade that student 4 received on quiz 3.*

Use Nested **for-loops** to Go Through a MDA

Example:

```
const int MAX1 = 10, MAX2 = 20;
int arr[MAX1][MAX2];  // MAX2 is called the “Base Dimension”
…
for (int i = 0; i < MAX1; i++)
{
   for (int j = 0; j < MAX2; j++)
   {
      cout << arr[i][j];    // Will print out all js for every i
   }
}
```

See demo file:
multidimensionalDemo.cpp

Initializing MDAs

- Recall that you can do this for uni-dimensional arrays and get all elements initialized to zero: `double numbers[100] = {0};`

- For multidimensional arrays, it's similar syntax:

```
double numbers[5][100] = { {0}, {0} };
```

OR:

```
double numbers[5][100] = {0};
```

- What would this do?

```
double numbers[2][3] = { {6,7}, {8,9} };
```

Multidimensional Array Parameters in Functions

- Recall that the size of an array is not needed when declaring a formal parameter:

```
void display_line(char a[ ], int size);
```

Look! No size!

Size is here instead!

- BUT the **base type must be completely specified in the parameter declaration** of a multi-dimensional array

```
void display_page(char page[ ][100], int size_dim1);
```

Only Base Dimension has a size defined!

</LECTURE>