Programming with Arrays
Intro to Pointers

CS 16: Solving Problems with Computers I
Lecture #11

Ziad Matni
Dept. of Computer Science, UCSB

MIDTERM IS COMING!

- **Thursday, 5/17** in this classroom
- **Starts at 2:00 PM **SHARP****
  - Please start arriving 5-10 minutes before class
- **I may ask you to change seats**
- Please bring your UCSB IDs with you

- **Closed book**: no calculators, no phones, no computers
- ***Only*** allowed ONE 8.5”x11” sheet of notes − one sided only
  - You have to turn it in with your exam
- **You will write your answers on the exam sheet itself.**

What's on the Midterm#2?
EVERYTHING *From Lectures 7 – 12, including…*

- Makefiles
- Debug Techniques
- Numerical Conversions
- Strings: C++ vs C-strings
- Strings and Characters: Member Functions & Manipulators
- File I/O
- Arrays
- Pointers

Lecture Outline

- Programming with Arrays
- Sequential Search of Arrays
- Multi-Dimensional Arrays

- Introduction to C++ Memory Map
- Introduction to Pointers

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 (DON’T WORRY ABOUT THIS UNTIL **AFTER** WE LEARN ABOUT POINTERS!)

Programming With Arrays

See demo file:
fillingUpArray.cpp

- The size requirement for an array might need to be **un-fixed**
  - Size is often *not known* when the program is written

- A common solution to the size problem
  (while still using "regular" arrays):
  - Declare the array size to be the **largest** that could be needed
  - Decide how to deal with *partially filled arrays*

Partially Filled Arrays

See demo file:
fillingUpArray.cpp

- When using arrays that are partially filled…
  - Functions dealing with the array may not need to know the **declared size of the array**
  - Only **how many maximum number of elements** need to be stored in the array!
- A parameter - let's call it **number_used** - may be sufficient to ensure that referenced index values are legal

Searching Arrays

- A **sequential search** is one way to search an array for a given value. The algorithm is as follows:

1. Look at each element from first to last to see if the target value is equal to any of the array elements

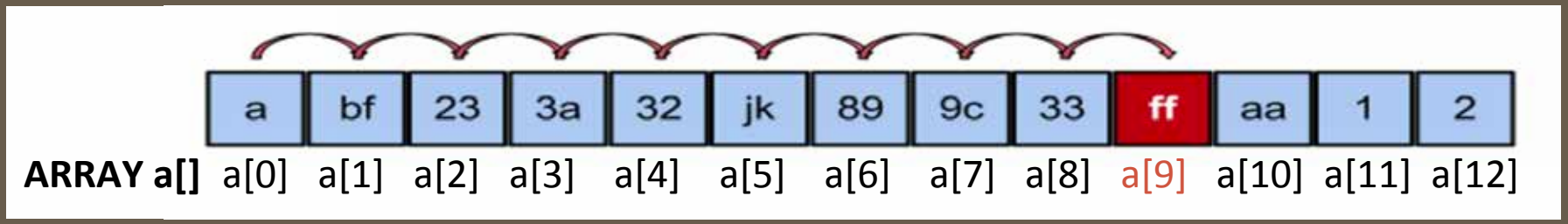
2. The index of the target value is returned to indicate where the value was found in the array

3. A value of -1 is returned if the value was not found anywhere

Pros? Cons?

Sequential Search

Task: Search the array for “ff”



Result: in position 9

```
int SeqSearch
(int arr[], int array_size, int target)
{
   int index(0);
   bool found(false);
   while ((!found) && (index < array_size))
   {
      if (arr[index] == target)
         found = true;
      else
         index++;
   }
   if (found)
      return index;
   else
      return -1;
}
```

Simple Sequential Search Function Example

1. *Look for a target value inside of a given array*
2. *If you find it, return its location (i.e. index) in the array*
3. *If you don't find it, return -1*

Searching an Array (*part 1 of 2*)

```
//Searches a partially filled array of nonnegative integers.
#include <iostream>
const int DECLARED_SIZE = 20;

void fill_array(int a[], int size, int& number_used);
//Precondition: size is the declared size of the array a.
//Postcondition: number_used is the number of values stored in a.
//a[0] through a[number_used-1] have been filled with
//nonnegative integers read from the keyboard.

int search(const int a[], int number_used, int target);
//Precondition: number_used is <= the declared size of a.
//Also, a[0] through a[number_used -1] have values.
//Returns the first index such that a[index] == target,
//provided there is such an index; otherwise, returns -1.

int main( )
{
    using namespace std;
    int arr[DECLARED_SIZE], list_size, target;

    fill_array(arr, DECLARED_SIZE, list_size);

    char ans;
    int result;
    do
    {
        cout << "Enter a number to search for: ";
        cin >> target;

        result = search(arr, list_size, target);
        if (result == -1)
            cout << target << " is not on the list.\n";
        else
            cout << target << " is stored in array position "
                 << result << endl
                 << "(Remember: The first position is 0.)\n";

        cout << "Search again?(y/n followed by Return): ";
        cin >> ans;
    }while ((ans != 'n') && (ans != 'N'));

    cout << "End of program.\n";
    return 0;
}
```

Searching an Array (*part 2 of 2*)

```
//Uses iostream:
void fill_array(int a[], int size, int& number_used)
<The rest of the definition of fill_array is given in Display 10.9.>

int search(const int a[], int number_used, int target)
{

    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;

    if (found)
        return index;
    else
        return -1;
}
```

Sample Dialogue

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 40
40 is stored in array position 3
(Remember: The first position is 0.)
Search again?(y/n followed by Return): y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return): n
End of program.
```

Multi-Dimensional Arrays

See demo file:
multidimensionalDemo.cpp

- 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

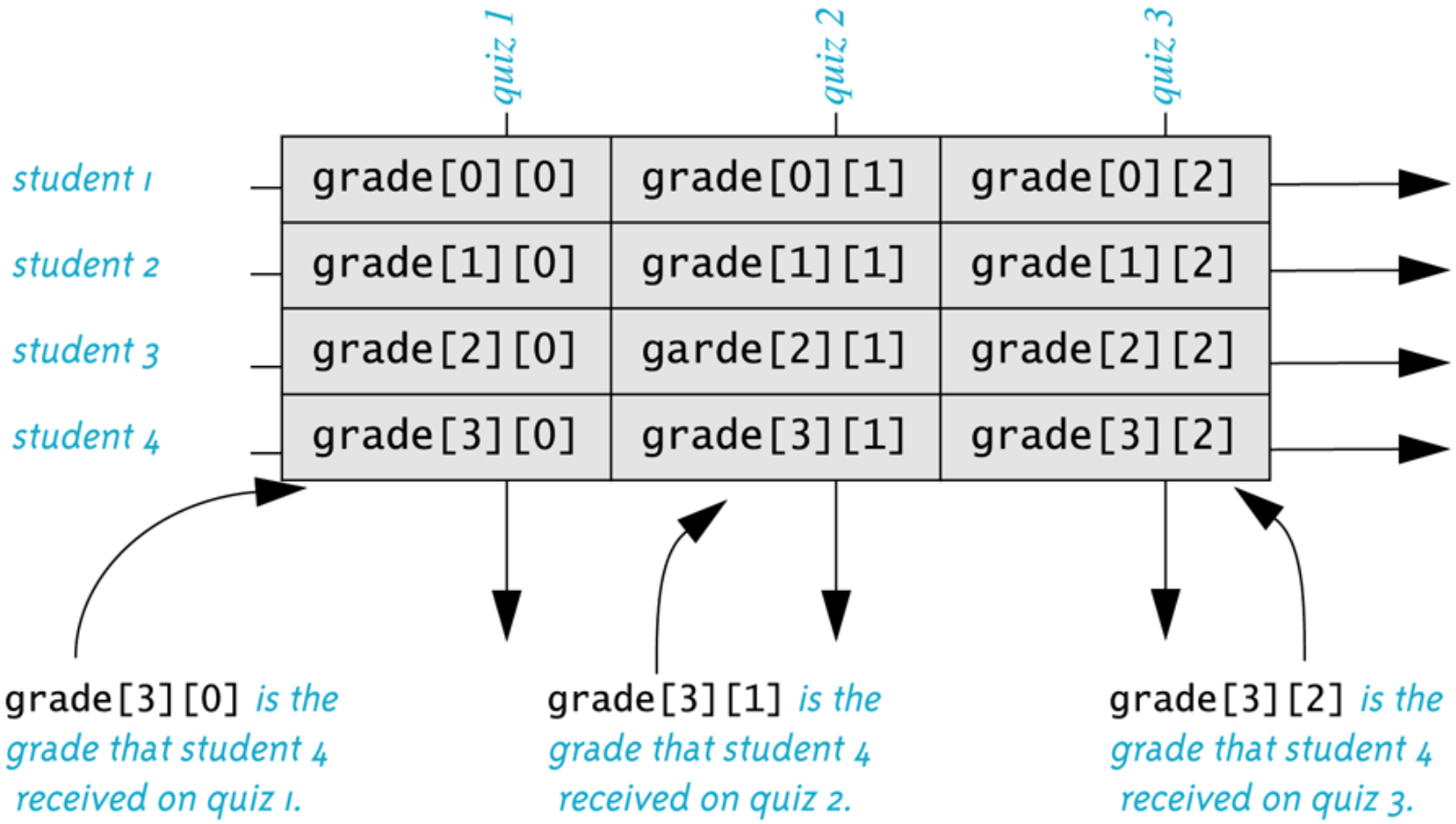
```
int grade[4][3];
```

  - The first array index refers to the number of a student
  - The second array index refers to a quiz number

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

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

The Two-Dimensional Array grade



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

Example:

```
const int MAX1 = 10, MAX2 = 20;
int arr[MAX1][MAX2];
…
for (int i = 0; i < MAX1; i++)
   for (int j = 0; j < MAX2; j++)
      cout << arr[i][j];
```

Initializing MDAs

See demo file:
multidimensionalDemo.cpp

- 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_dimension1);
```

Base has a size defined!

INTRO TO POINTERS

Section 9.1 in book

Passing by Values

What does this code print out?

```
void swapValue(int x, int y)
{
   int tmp = x;
   x = y;
   y = tmp;
}
int main()
{
   int a(30), b(40);
   cout << a << " " << b << endl;
   swapValue(a, b);
   cout << a << " " << b << endl;
}
```

Is It:

A.

30 40
30 40

B.

30 40
40 30

C.

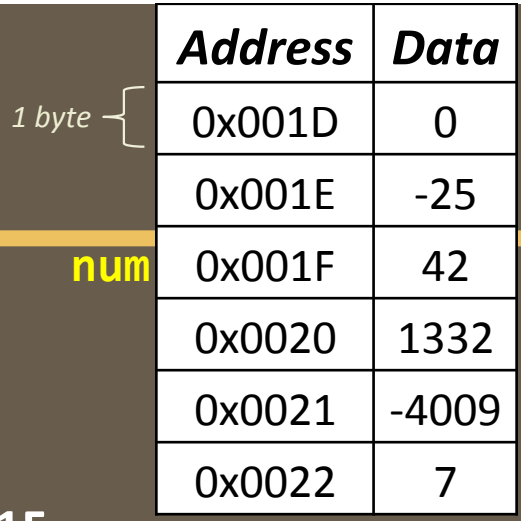
Something else

Pointers

- A pointer is the **memory address** of a variable

- When a variable is used as a call-by-reference argument, it's the actual address in memory that is passed

Memory Addresses



- Consider int variable **num** that holds the int value **42**
- **num** is assigned a place in memory (what does that??)

- In this example the ***address*** of that place in memory is **0x001F**
  - Generally, memory addresses use *hexadecimals*
    (and usually 8 of them, not just 4… but this is ONLY an example…)
  - The "**0x**" at the start is just to indicate the number is a hexadecimal

Memory Addresses

- The address of a variable can be obtained by putting the ampersand character (&) before the variable name.
- **&** is called the ***address-of*** operator
- Example:
  `int num_add = &num;`
  will result in **num_add** to hold the value **0x001F** (or 31 in decimal)

1 byte

	Address	***Data***
	0x001D	0
	0x001E	-25
num	0x001F	42
	0x0020	1332
	0x0021	-4009
	0x0022	7
	…	…
num_add	0x98A0	31

YOUR TO-DOs

- ❑ Turn in Lab 6 on Monday
- ❑ Do HW11 by next Tuesday
- ❑ Study for your Midterm #2 on Thursday!

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

- ❑ Enjoy the beautiful outdoors

</LECTURE>