CMSC202
Computer Science II for Majors

Lecture 03 –
Arrays and Functions

Dr. Katherine Gibson

Last Class We Covered

- C++ Primer
  - Arithmetic expressions
  - Operators
  - Type casting
  - Input / Output
  - C-Strings
  - Control Structures

Any Questions from Last Time?

Today's Objectives

- To understand how functions work in C++
  - Prototype
  - Definition
  - Call
- To cover the basics of arrays
  - Declaration and initialization
  - Multi-dimensional arrays
- To examine how arrays and functions interact

Functions

"Parts" of Using Functions

- Function Prototype (Declaration)
  - Information for compiler
  - To properly interpret calls
- Function Definition
  - Actual implementation (i.e., code) for function
- Function Call
  - How function is actually used by program
  - Transfers control to the function

Function Prototype

- Gives compiler information about the function
  - How to interpret calls to the function

```
<return type> <fxn name> (<parameters>);
int squareNumber (int n);
```

  - Must have the parameter's data types

- Placed before any calls
  - In declaration space of **`main()`**
  - Or above **`main()`** for global access

Function Definition

- Implementation of the function
  - Just like implementing the **`main()`** function

```
int squareNumber (int n) {
    int answer = n * n;
    return answer;
}
```

- Function definition must match prototype

Function Definition Placement

- Placed after the function **`main()`**
  - NOT inside the function **`main()`** !
- All functions are equal
  - No function is contained inside another

- Function name, parameter type, and return type all must match the prototype's
- **`return`** statement sends data back to the caller

Function Call

- Very similar to how it's done in Python

```
int tenSquared = squareNum(10);
```

- **`squareNum()`** returns an integer
  - Assigned to the variable **`tenSquared`**
- Arguments: the "literal" 10
  - Could also have passed a variable
  - Must be of type **`int`** – why?

Function Example (part 1)

Display 3.5

```
1   #include <iostream>
2   using namespace std;

3   double totalCost(int numberParameter, double priceParameter);
4   //Computes the total cost, including 5% sales tax,
5   //on numberParameter items at a cost of priceParameter each.

6   int main( )
7   {
8       double price, bill;
9       int number;

10      cout << "Enter the number of items purchased: ";
11      cin >> number;
12      cout << "Enter the price per item $";
13      cin >> price;

14      bill = totalCost(number, price);
```

Function declaration; also called the function prototype

Function call

Function Example (part 2)

```
15      cout.setf(ios::fixed);
16      cout.setf(ios::showpoint);
17      cout.precision(2);
18      cout << number << " items at "
19           << "$" << price << " each.\n"
20           << "Final bill, including tax, is $" << bill
21           << endl;

22      return 0;
23  }

24  double totalCost(int numberParameter, double priceParameter)
25  {
26      const double TAXRATE = 0.05; //5% sales tax
27      double subtotal;

28      subtotal = priceParameter * numberParameter;
29      return (subtotal + subtotal*TAXRATE);
30  }
```

Function head

Function body

Function definition

Function Example (usage)

- Notice that the formatting changes made to **`cout`** are persistent
- They applied to both $10.10 and $21.21

SAMPLE DIALOGUE

```
Enter the number of items purchased: 2
Enter the price per item: $10.10
2 items at    $10.10 each.
Final bill, including tax, is $21.21
```

Parameters vs Arguments

- Often used interchangeably
  - Technically, parameter is formal variable name; argument is actual value or variable

- "Formal" parameters/arguments
  - In the function prototype
  - In the function definition
- "Actual" parameters/arguments
  - In the function call

Returning "void"

- "void" functions are for when we don't need to get a value back from the function
  - Used for functions that only have side effects
  - (e.g., printing out information)

- Returning "**void**" means no value is returned
- Declaration is similar to "regular" functions

```
void printResults(double cost, double tax);
```

  - Nothing is returned

"`return`" Statements

- Transfers control back to the calling function
- "`return`" statement optional for void functions
- All other returns types must have a return statement in the function
  - An error results otherwise
- Typically the last statement in the definition
  - Why?

Pre- and Post-Conditions

- For this class, you'll need to include function headers in your code (coding standards)
  - Must contain name, pre-, and post-conditions

- Conditions include assumptions about program state, not just the input and output

```
// Function name: showInterest
// Pre-condition: balance is nonnegative account
//    balance; rate is interest rate as percentage
// Post-condition: amount of interest on given
//    balance, at given rate
void showInterest(double balance, double rate);
```

Library Functions

- C++ has libraries full of useful functions!
- Must "**`#include`**" appropriate library
  - e.g.,
  - **`<cmath>`**, **`<cstdlib>`** (Original "C" libraries)
  - **`<iostream>`** (for cout, cin)

- Library functions are used in the same way as programmer-created functions

Useful Library Functions (part 1)

Display 3.2 Some Predefined Functions

NAME	DESCRIPTION	TYPE OF ARGUMENTS	TYPE OF VALUE RETURNED	EXAMPLE	VALUE	LIBRARY HEADER
sqrt	Square root	double	double	sqrt(4.0)	2.0	cmath
pow	Powers	double	double	pow(2.0,3.0)	8.0	cmath
abs	Absolute value for int	int	int	abs(-7) abs(7)	7 7	cstdlib
labs	Absolute value for long	long	long	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	Absolute value for double	double	double	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath

Useful Library Functions (part 2)

ceil	Ceiling (round up)	double	double	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	Floor (round down)	double	double	floor(3.2) floor(3.9)	3.0 3.0	cmath
exit	End pro-gram	int	void	exit(1);	None	cstdlib
rand	Random number	None	int	rand( )	Varies	cstdlib
srand	Set seed for rand	unsigned int	void	srand(42);	None	cstdlib

The `main()` Function

- **`main()`** is also a function – a "special" one!
- One (and only one) function called **`main()`** can exist in a program
- The operating system calls **`main()`**
  - Not the programmer!
- Should return an integer (0 is traditional)
  - `return 0;`

Why Use Functions?

- Allows you to build "blocks" of programs
  - Divide and conquer large problems
- Increases readability and reusability
  - Put in a separate file from **`main()`** for easy sharing

- Quick note:
  - **Functions in C++ can only return <u>one</u> thing!!!**
  - (For now)

Arrays

Arrays

- An array is a collection of related data items
  - An array can be of any data type you choose
  - They all have the same data type

- Arrays are static – their size does not change
  - They are declared *contiguously* in memory
  - In other words, an array's data is stored in one big block, together

Declaring Arrays

- Declaration:

```
<type> <name> [size];
float  xArray [10];
```

  – This array now has room to hold 10 floats

- Indexing starts at 0:

```
xArray[9]; /* end of the array */
```

Limitations of an Array

- An array does not know how large it is
  - No **`size()`** function
  - No bounds checking – you must be careful!

- Arrays are static
  - Their size must be known at compile time
  - Their size cannot be changed once set
  - You cannot use user input for array size

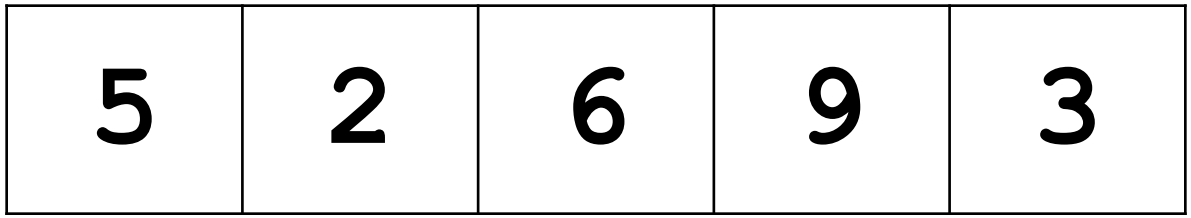
    ~~"How many numbers would you like to store?"~~

Array Initialization

- A declaration does not initialize the data stored in the memory locations
  - They contain "garbage" data
    - Whatever is left from the last time it was used

- An array can be initialized at declare time:

```
int numbers[5] = { 5, 2, 6, 9, 3 };
```



Auto-Initialization

- If there are fewer values than the given size:
  - Fills values starting at the beginning
  - Remainder is filled with that data type's "zero"

- If no array size is given:
  - Array is created only as big as is needed based on the number of initialization values

```
int yArray[] = {5, 12, 11};
```

  - Allocates array **yArray** to hold 3 integers

C-String Initialization

- C-Strings are *arrays* of characters

- They can be initialized in the normal way

```
char name[5] = {'J', 'o', 'h', 'n', 0 };
```

- Or with a string constant:

```
char name[5] = "John";
```

- Different quotes have different purposes!!!
  - Double quotes are for strings
  - Single quotes are for chars (characters)

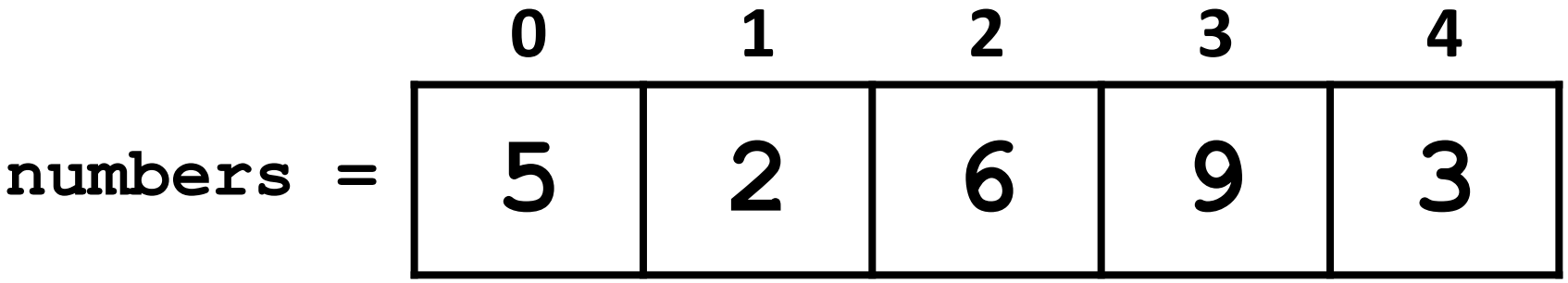
Accessing Array Elements

- Use square brackets to access a single element

```
cout << "The third element is "
     << numbers[2] << endl;
```

- This gives the output:

```
The third element is 6
```



Accessing Array Elements

- C++ also accepts any expression as a "subscript"
  - But it must evaluate to an integer

```
numbers[(start + end) / 2];
```

- **IMPORTANT!**
- C++ does not do bounds checking for simple arrays, so you must ensure you are staying within bounds

Constants as Array Size

- You should always used a defined/named constant for your array size
  - Do not use magic numbers!

```
const int NUMBER_OF_STUDENTS = 5;
int score[NUMBER_OF_STUDENTS];
```

- Improves readability, versatility, and maintainability

Arrays and Functions

Arrays in Functions

- As arguments to functions
  - Indexed variables
    - An individual element of an array can be passed
  - Entire arrays
    - All array elements can be passed as one entity
- As return value from function
  - Can be done, but we'll cover it later

Passing Indexed Variables

- Handled the same way as a "regular" variable
  - Function declaration:

```
void myFunction(double par1);
```

  - Variables:

```
double n, a[10];
```

  - Valid function calls:

```
myFunction(a[3]); // a[3] is double
myFunction(n);    // n is double
```

Passing Entire Arrays

- Formal parameter can be an entire array
  - Passed into the function by the array's name
  - Called an array parameter

- Must send size of array as well
  - Typically done as second parameter
  - Simple integer-type formal parameter

Live Coding Exercise

`fillUp.cpp`

Arrays, initialization, and functions

Array Parameter Example

Display 5.3 Function with an Array Parameter

SAMPLE DIALOGUEFUNCTION DECLARATION

```
void fillUp(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.
```

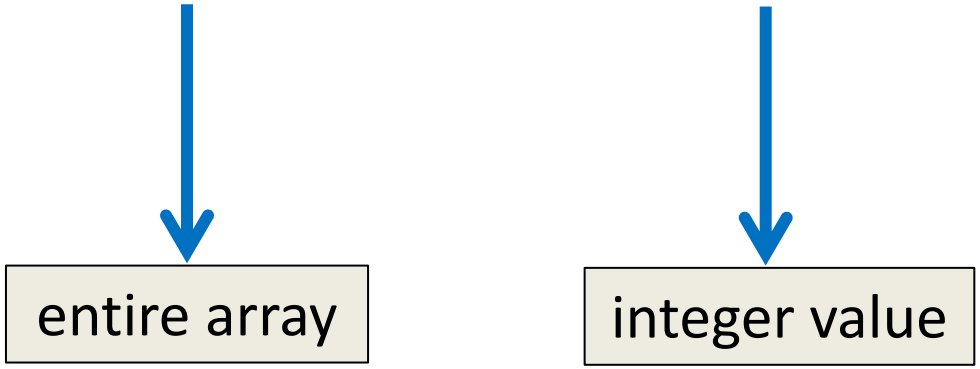
SAMPLE DIALOGUEFUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size - 1) << endl;
}
```

Array as Argument Example

- Using the function from the previous slide, consider this code, inside a **`main()`** function:

```
int score[5], numberOfScores = 5;
fillUp(score, numberOfScores);
```



Note the lack of `[]` brackets in the array argument!

Array as Argument Explanation

- How does this work? What's being passed?
- Think of an array as 3 components:
  - Address of first indexed variable (**`arrName[0]`**)
  - Array base type
  - Size of array
- Only the first piece is passed!
  - Just the beginning address of the array
  - We'll discuss this in detail later

Array Parameters

- Array size must be sent separately
  - But this actually increases functionality!

- Same function can be used on any size array!

```
int score[5], time[10];
fillUp(score, 5);
fillUp(time, 10);
```

Multidimensional Arrays

- Arrays with more than one index

```
char page[30][100];
```

  - Two indices: an "array of arrays"
  - Visualize as:

```
page[0][0],  page[0][1],  ..., page[0][99]
page[1][0],  page[1][1],  ..., page[1][99]
...
page[29][0], page[29][1], ..., page[29][99]
```

- C++ allows any number of indexes
  - Typically no more than two or three

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter of function
  - 2nd dimension size is given

```
void DisplayPage(char page[][100], int numRows) {
    for ( int i = 0; i < numRows; i++ ) {
        for ( int j = 0; j < 100; j++ ) {
            cout << page[i][j];
        }
        cout << endl;
    }
}
```

Review: Array Limitations

- Simple arrays have limitations
  - Array out-of-bounds access
  - No resizing
  - Hard to get current size
  - Not initialized
  - Mostly due to efficiency and backwards-compatibility, which are high priorities in C/C++
- Later, we will learn about the vector class, which addresses many of these issues

Announcements

- The course policy agreement is due back in class by Tuesday, February 9th
  - Worth 1% of your grade
  - (Final is now worth 19%)
- The Blackboard site is now available
  - Course schedule is now available
- Next time: Pointers and References