Arrays

CS 16: Solving Problems with Computers I
Lecture #13

Ziad Matni
Dept. of Computer Science, UCSB

Announcements

- New Homework #12 given today (due Thursday)
- New Lab #7 given on Wednesday

Lecture Outline

Ch.7 Arrays

- Arrays in Functions
- Algorithmic Designs with Arrays

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;
```

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 [ ]

- In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces { }

- So, the example makes the array **num** automatically size 3.

- 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**, that 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[] );
```

Arrays as Function Arguments

- You *can* make an entire array a formal parameter for a function
- But you *cannot* make an entire array the RETURNED value for a function

- An array parameter
  - is not a call-by-value parameter
  - is not a **call-by-reference** parameter
  - Although it **behaves** much like call-by-reference parameters

Array Parameter Declaration

- 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 [ ] !!!)

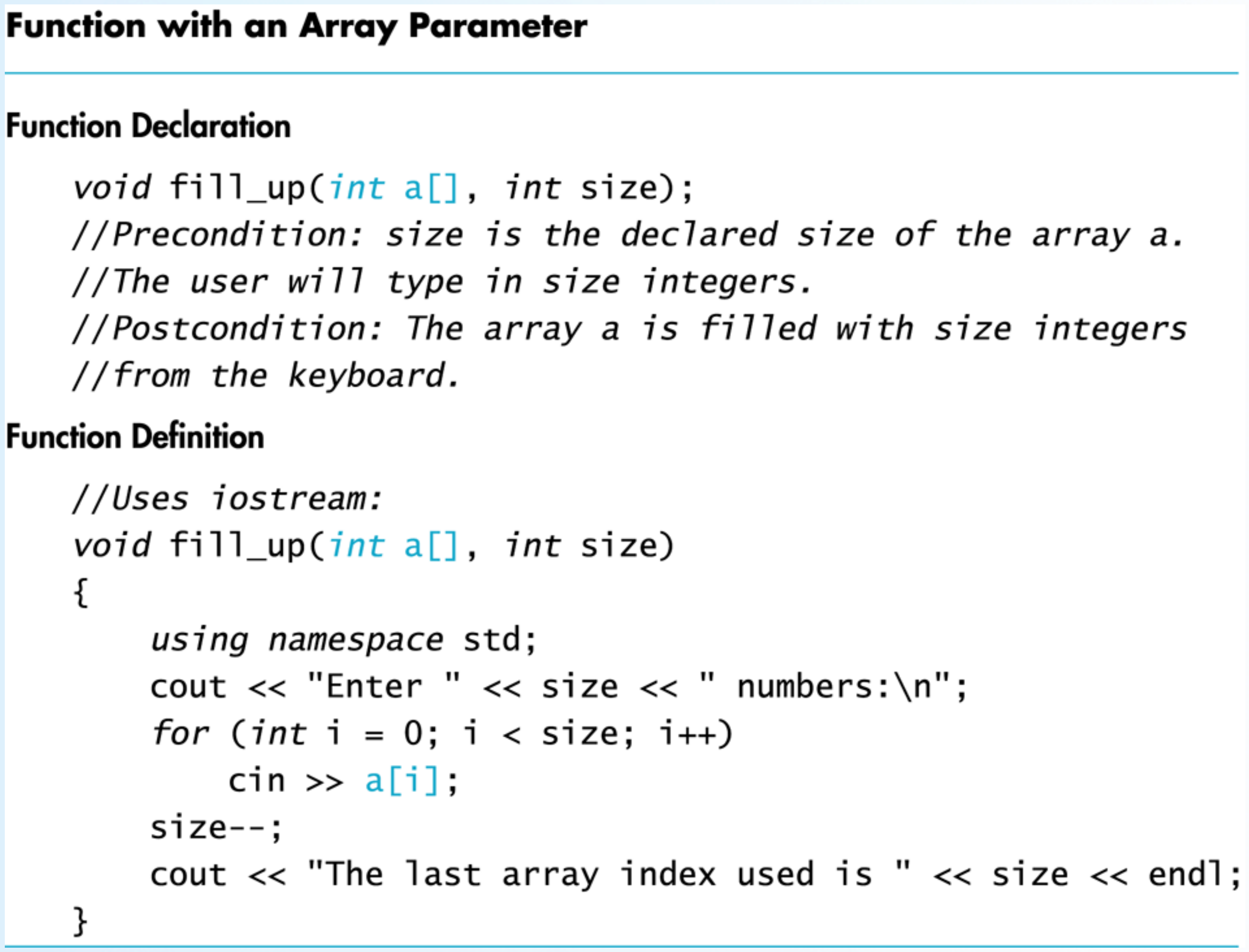
```
void fill_up(int a[], int size);
```

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

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

- **fill_up** is *called* in this way (no [ ] !!!)

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



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

- 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 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
    - 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

- 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
- 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 a class type

- **Functions cannot return arrays**

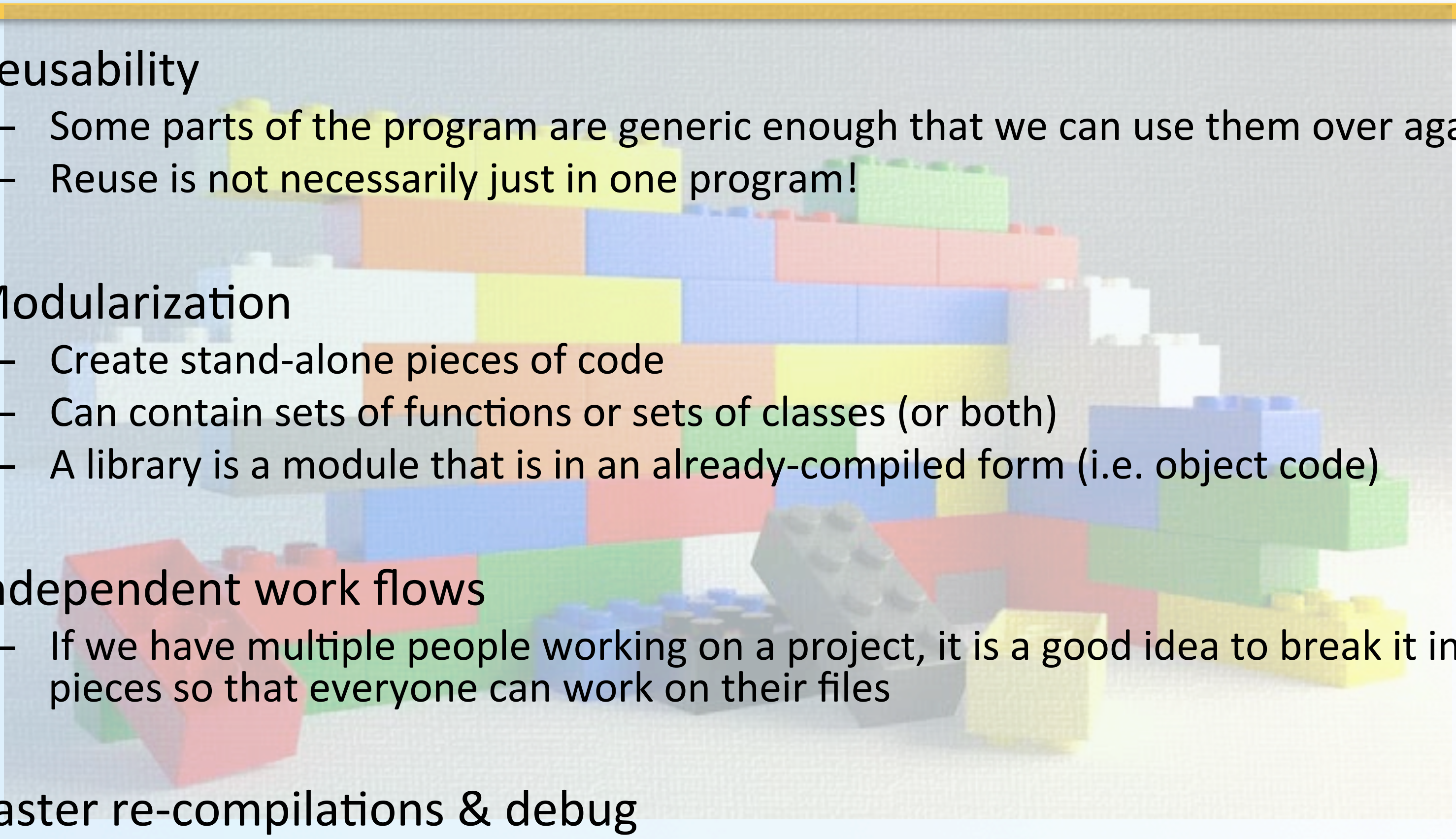
- We learn later how to return
  a pointer to an array

C++ Programming in Multiple Files

- Novice C++ Programming:
  - All in one .cpp source code file
  - All the function definitions, plus the main( ) program

- Actual C++ Programming separates parts
  - There are usually one or more ***header files*** with file names ending in **.h** that typically contain function prototypes
  - There are one or more files that contain function definitions, some with **main( )** functions, and others that don't contain a **main( )** function

Why?



- Reusability
  - Some parts of the program are generic enough that we can use them over again
  - Reuse is not necessarily just in one program!
- Modularization
  - Create stand-alone pieces of code
  - Can contain sets of functions or sets of classes (or both)
  - A library is a module that is in an already-compiled form (i.e. object code)
- Independent work flows
  - If we have multiple people working on a project, it is a good idea to break it into pieces so that everyone can work on their files
- Faster re-compilations & debug
  - When you make a change, you only have to re-compile the part(s) that have changed
  - Easier to debug a portion than the entire program!

```
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);




float linearScale(...)
{ ... }
float quadraticScale(...)
{ ... }
float bellCurve(...) { ... }




int main()
{
    ...
}
```

```
// File: MyFunctions.h
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);

// File: MyFunctions.cpp
#include "MyFunctions.h"
#include <iostream>, etc…

float linearScale(...)
{ ... }
float quadraticScale(...)
{ ... }
float bellCurve(...) { ... }

// File: main.cpp
#include "MyFunctions.cpp"
#include <iostream>, etc…

int main()
{
    ...
}
```

Compiling Everything…

```
g++ -c MyFunctions.cpp –o Myfunctions.o
```

(creates MyFunctions.o)

```
g++ -c main.cpp –o main.o
```

(creates main.o)

The –c option creates object code – this is machine language code, but it's not the entire program yet… The target object file here is always generated as a ***.o*** *type*

```
g++ -o ProgX main.o MyFunctions.o
```

(creates ProgX)

The –o option creates object code – in this case, it's object code created from other object code. The result is the entire program in executable form. The object file here is always generated with the name specified after the –o option.

What Do You End Up With?

MyFunctions.h	Header file w/ function prototypes
MyFunctions.cpp	C++ file w/ function definitions
MyFunctions.o	Object file of MyFunctions.cpp
main.cpp	C++ file w/ main function
main.o	Object file of main.cpp
ProgX	"Final" executable file

…and this is a simple example!!…

Wouldn't it be nice to have code that generates/controls this?

Make to Tie Them All Together

- "Make" is a *build automation tool*
  - Automatically builds executable programs and libraries from source code
  - The instructions for **make** is in a file called ***Makefile***.

- Makefile is code written in OS-friendly code

Makefile

- The file must be called "makefile" (or "Makefile")
- Put all the instructions you're going to use in there
  - Just type "make" at the prompt, instead of all the g++ commands
- Makefiles can easily be used to do other OS-related stuff
  - Like "clean up" your area, for example

Syntax of a Make

Target "all" programs in this project

dependencies

recipe lines

Target something specific

Doesn't have to be compiling only!

```
all: MyFunctions.cpp main.cpp
        g++ -std=c++11 –Wall -c MyFunctions.cpp
        g++ -std=c++11 –Wall -c main.cpp
        g++ -std=c++11 –Wall -o Progx main.o MyFunctions.o

ProgY: MyOtherFunctions.cpp
        g++ -std=c++11 –Wall -c MyOtherFunctions.cpp –o ProgY

clean:
        rm *.o ProgX ProgY
```

Syntax of a Make

Target "all" programs in this project

Dependencies that are declared below

```
all: Exercise1 Exercise2

Exercise1: ex1.cpp
        g++ -std=c++11 –Wall –c ex1.cpp –o ex1

Exercise2: ex2.cpp
        g++ -std=c++11 –Wall –c ex2.cpp –o ex2
```

To Dos

- Homework #12 due Thursday

</LECTURE>