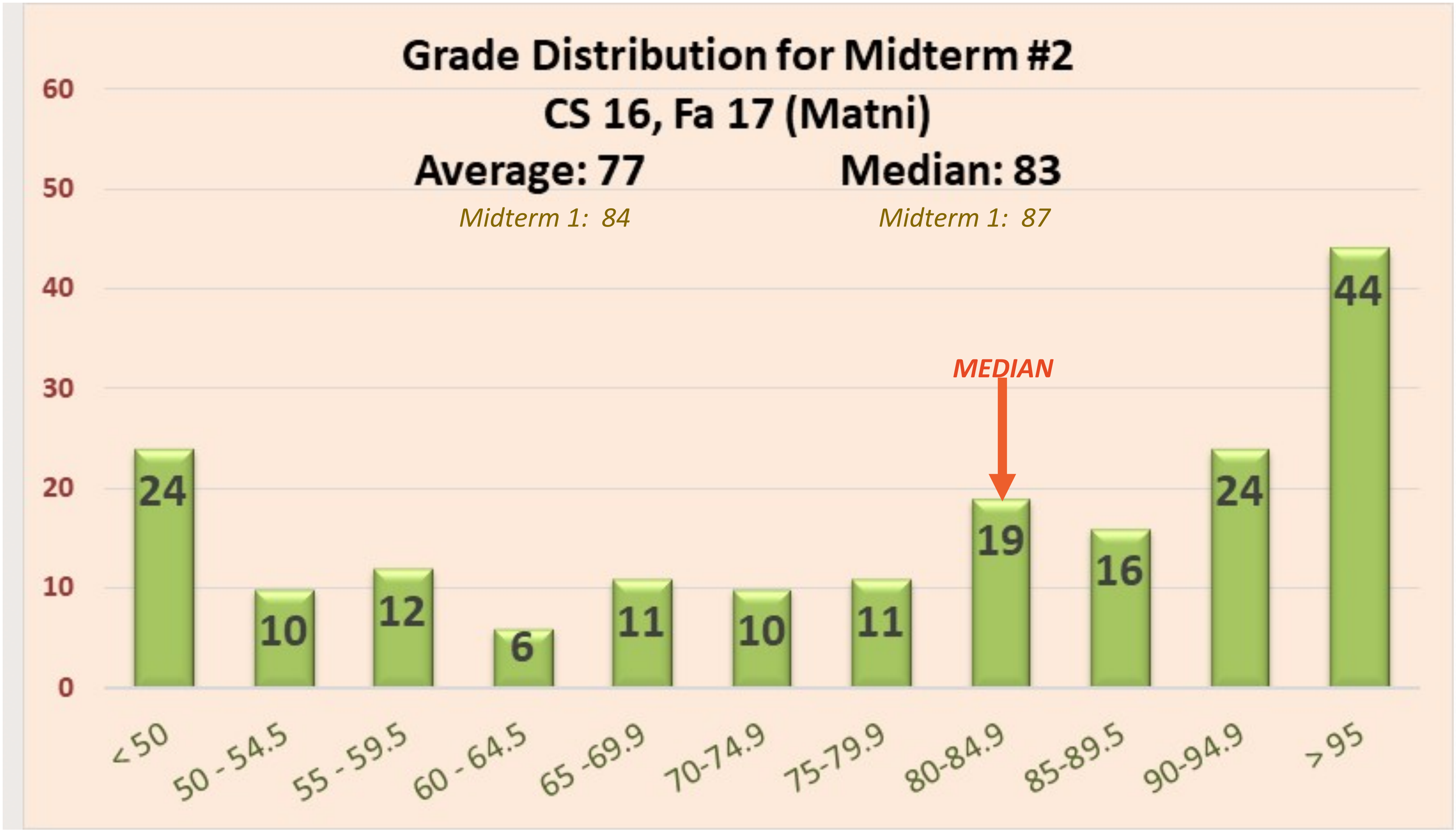
Dynamic Arrays
Makefiles and Multiple File Compiles

CS 16: Solving Problems with Computers I
Lecture #14

Ziad Matni
Dept. of Computer Science, UCSB

WHAT THE NEXT 3 WEEKS LOOK LIKE

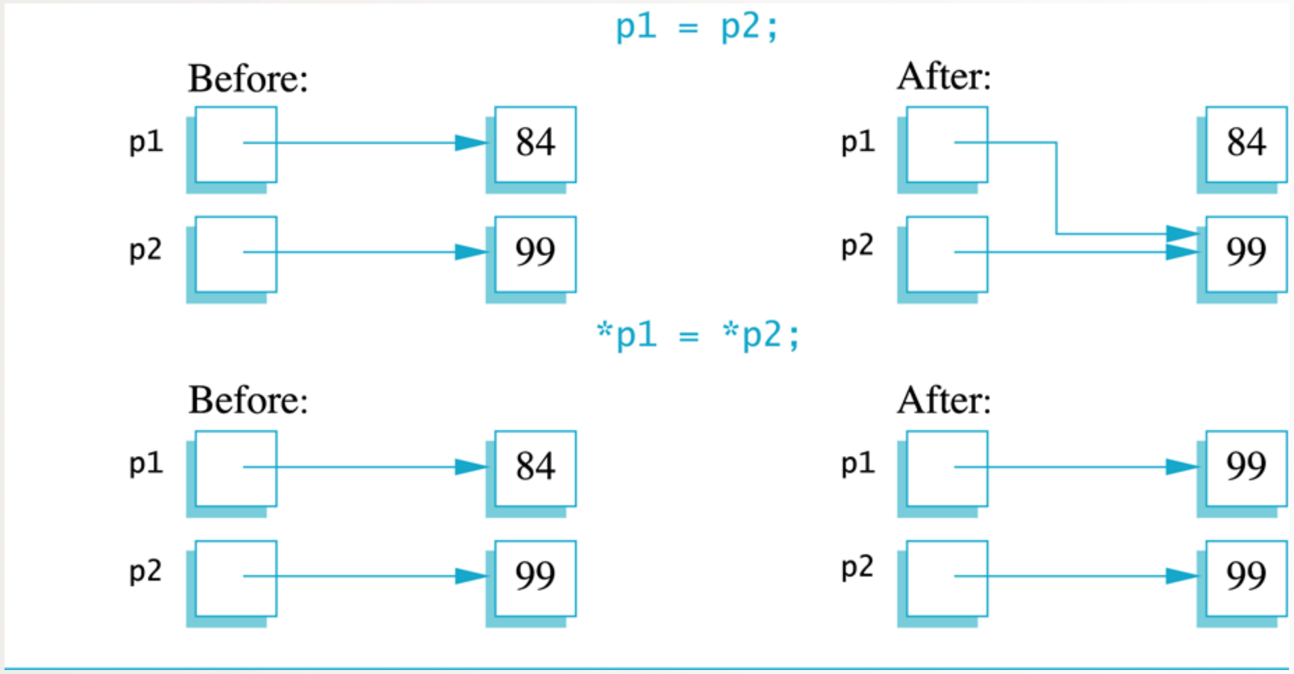
MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY
20-Nov	21-Nov	22-Nov	23-Nov	24-Nov
Lab 8 issued	***Lecture:*** Dynamic Arrays, Makefiles		▨	▨
			▨	▨
			▨	▨
27-Nov	28-Nov	29-Nov	30-Nov	1-Dec
Lab 9 issued	***Lecture:*** Structures and Classes	Lab8 due	***Lecture:*** Linked Lists	
	Hw 8 due			
	Hw 9 issued			
4-Dec	5-Dec	6-Dec	7-Dec	8-Dec
Lab attendance is optional	***Lecture:*** Recursion, Search/Sort	Lab 9 due	***Lecture:*** Review for Final Exam	
			Hw 9 due	
11-Dec	12-Dec	13-Dec	14-Dec	15-Dec
	FINAL EXAM, 4 - 7 PM			



Grade Distribution for Midterm #2
CS 16, Fa 17 (Matni)
Average: 77
Median: 83
Midterm 1: 84
Midterm 1: 87
MEDIAN
60
50
40
30
20
10
0
24
10
12
6
11
10
11
19
16
24
44
< 50
50 - 54.5
55 - 59.5
60 - 64.5
65 -69.9
70-74.9
75-79.9
80-84.9
85-89.5
90-94.9
> 95

Refresher on Pointers

Uses of the Assignment Operator on Pointers



2 Main Ways to Define Pointers

```
int *ptr, num;
…
num = 5;
ptr = &num;
// ptr points to num
…
cout << *ptr;
// shows 5
```

```
int *ptr;
ptr = new int;
…
*ptr = 5;
// points to a place in the heap
…
cout << *ptr;
// shows 5

delete ptr;
// remove from the heap
```

Type Definitions

- A name can be assigned to a type definition, then used to declare variables

- The keyword **typedef** is used to define new type names
- Syntax:

typedef *Known_Type_Definition* New_Type_Name;

example:

```
typedef int* MyintPtr;
```

Defining Pointer Types

- This helps to avoid mistakes using pointers:
- Example: `typedef int* IntPtr;`

Defines a new custom *data type*, **`IntPtr`**,

for pointer variables containing pointers to **`int`** variables

`IntPtr p;`

is now equivalent to saying: `int *p;`

Pointer Reference Parameters

- An advantage in using **typedef** to define a pointer type is seen in *call-by-reference* parameter lists, like…

- Example:

```
    void sample_function(IntPtr& pointer_var);
```

is less confusing than:

```
    void sample_function(int*& pointer_var);
```

Dynamic Arrays

*Read Ch. **9** (Pointers) in textbook*

Dynamic Arrays

A dynamic array is an array whose size is determined when the program is running, not when you write the program

Is a vector a dynamic array?

Pointer Variables and Array Variables

- Array variables are *actually* **pointer variables** that point to the first indexed variable!
  - Remember when calling an array in a function?
    - funcA(a) … not … funcA(a[ ])

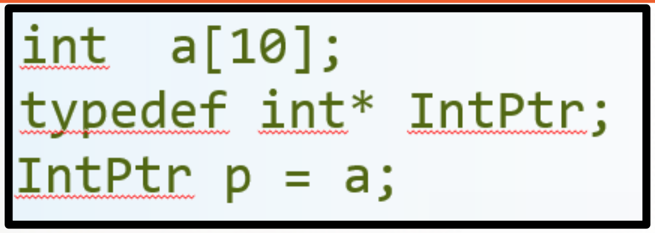
Since **a** is a pointer variable that points to **a[0]**, then issuing: **p = a;** causes **p** to point to the same location as **a**

- Take, for instance:

```
int  a[10];
typedef int* IntPtr;
IntPtr p;
```

*NOTE: Variables **a** and **p** are **the same kind of variable!***

Pointer Variables *As* Array Variables

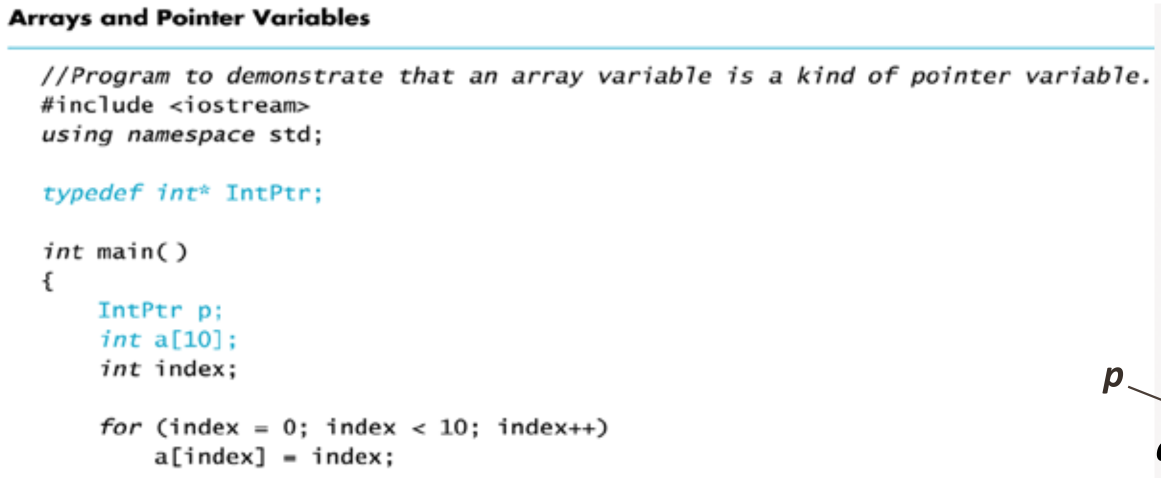


```
int  a[10];
typedef int* IntPtr;
IntPtr p = a;
```

- Continuing with the previous example:
  Pointer variable **p** can be used
  as if it were an array variable!!
- So, **p[0]**, **p[1]**, …**p[9]** are all legal ways to use **p**

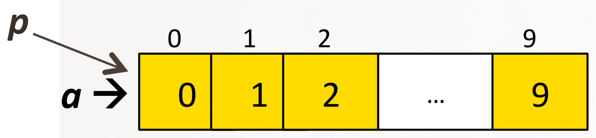
- *Is there a difference between an array and a pointer?*
  Variable **a** can be used as a pointer variable BUT the pointer value in **a** cannot be changed
  – So, the following is **<u>not</u>** legal:

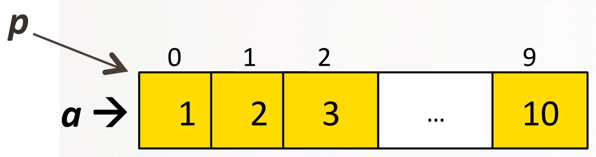
```
IntPtr p2;    // let's say p2 is assigned a value
a = p2        // attempt to change a is NOT OK!
```



Arrays and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;
```





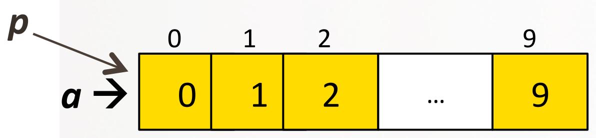
Arrays and Pointer Variables

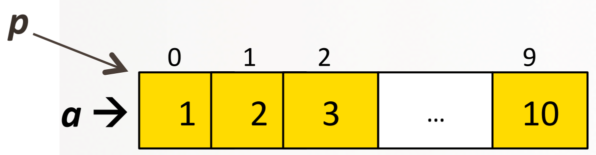
```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p are also changes to the array a.

Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```





Creating Dynamic Arrays

- Normal arrays require that the programmer determine the size of the array ***when the program is written***
  - *What if the programmer estimates too large?*
    - Memory is wasted
  - *What if the programmer estimates too small?*
    - The program may not work in some situations

- Dynamic arrays can be created with just the right size ***while the program is running***

Are Dynamic Arrays *aka* Vectors?!

- Not exactly the same…
  - ***vector*** is one *implementation* of dynamic arrays
  - “dynamic arrays” is a bigger (more encompassing) term

- The biggest difference is:
  - Vectors ***automatically*** increase their capacity
  - Dynamic arrays have to be told to do this using **new** and **delete**

- The advantage of vectors is that they are well-defined and you don’t have to worry about size changes, capacity adjustments in memory, etc…

Creating Dynamic Arrays

- Dynamic arrays are created using the **new** operator

- Example:
  To create an array of 10 elements of type double:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

d can now be used as if it were an ordinary array!

Dynamic Arrays (cont.)

- Pointer variable d is a pointer to d[0]
- When finished with the array, it should be **delete**d to return memory to the **heap (freestore)**
  - Example showing syntax: `delete [ ] d;`
  - The brackets tell C++ that a dynamic array is being deleted so it must check the size to know how many indexed variables to remove
  - Do not forget the brackets!
- Display 9.6 in the book has an example of use

Multidimensional Dynamic Arrays

- Example: **Create a 3x4 multidimensional dynamic array**
- Recall: multidimensional arrays are arrays of arrays…
  - So a 3x4 array = 3-element array, each of which is a 4-element array
- First create a one-dimensional dynamic array
  - Start with a new definition:

    ```
    typedef int* IntArrayPtr;
    ```
  - Now create a dynamic array of pointers named **m**:

    ```
    IntArrayPtr m = new IntArrayPtr[3];
    ```

Multidimensional Dynamic Arrays

- First create a one-dimensional dynamic array
  - Start with a new definition:

```
typedef int* IntArrayPtr;
```

- Now create a dynamic array of pointers named m:

```
IntArrayPtr m = new IntArrayPtr[3];
```

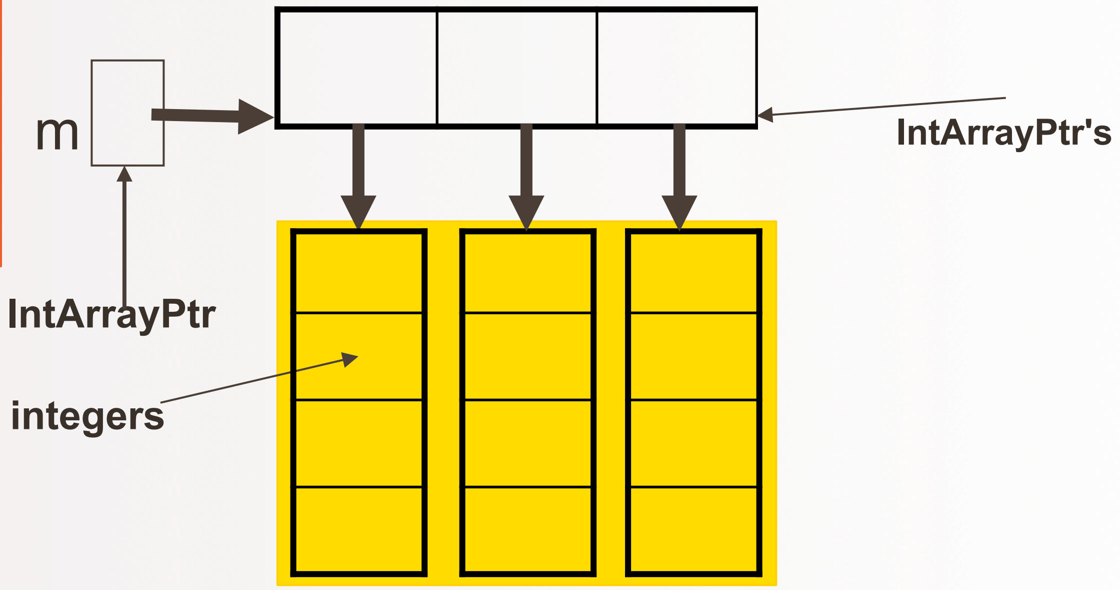
- For each pointer in **m**, create a dynamic array of integers

```
for (int i = 0; i < 3; i++)
        m[i] = new int[4];
```

A Multidimensional Dynamic Array

The dynamic array created on the previous slide could be visualized like this:

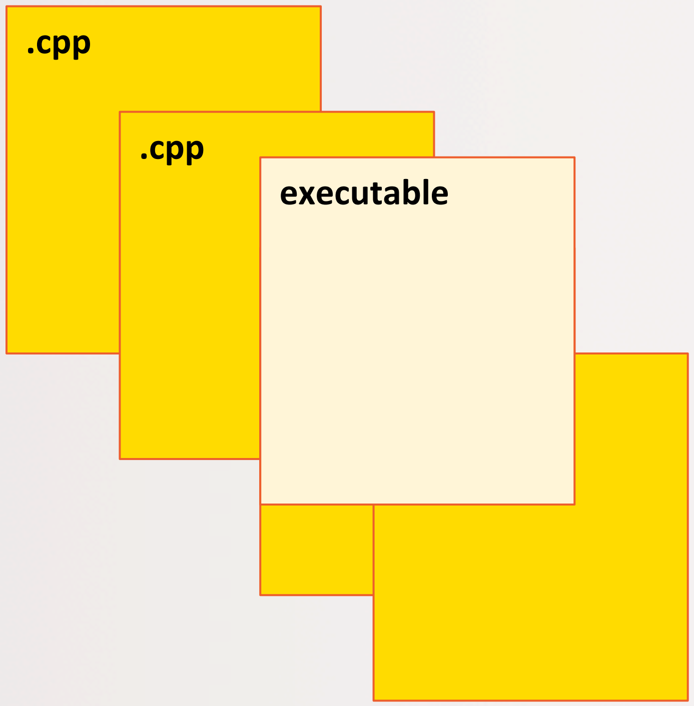
```
typedef int* IntArrayPtr;
IntArrayPtr m =
         new IntArrayPtr[3];
for (int i = 0; i < 3; i++)
   m[i] = new int[4];
```



What do you do After You're Done with a MDDA?

- To **delete** a multidimensional dynamic array
  - Each call to **new** that created an array must have a corresponding call to **delete[ ]**
  - Example: To delete the dynamic array created on the previous slide:

```
for ( i = 0; i < 3; i++)
    delete []m[i];      // delete the arrays of 4 int's
delete []m;             // delete the array of IntArrayPtr's
```



MULTIPLE FILE COMPILATIONS

C++ Programming in Multiple Files

- Novice C++ Programming:
  - All code 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

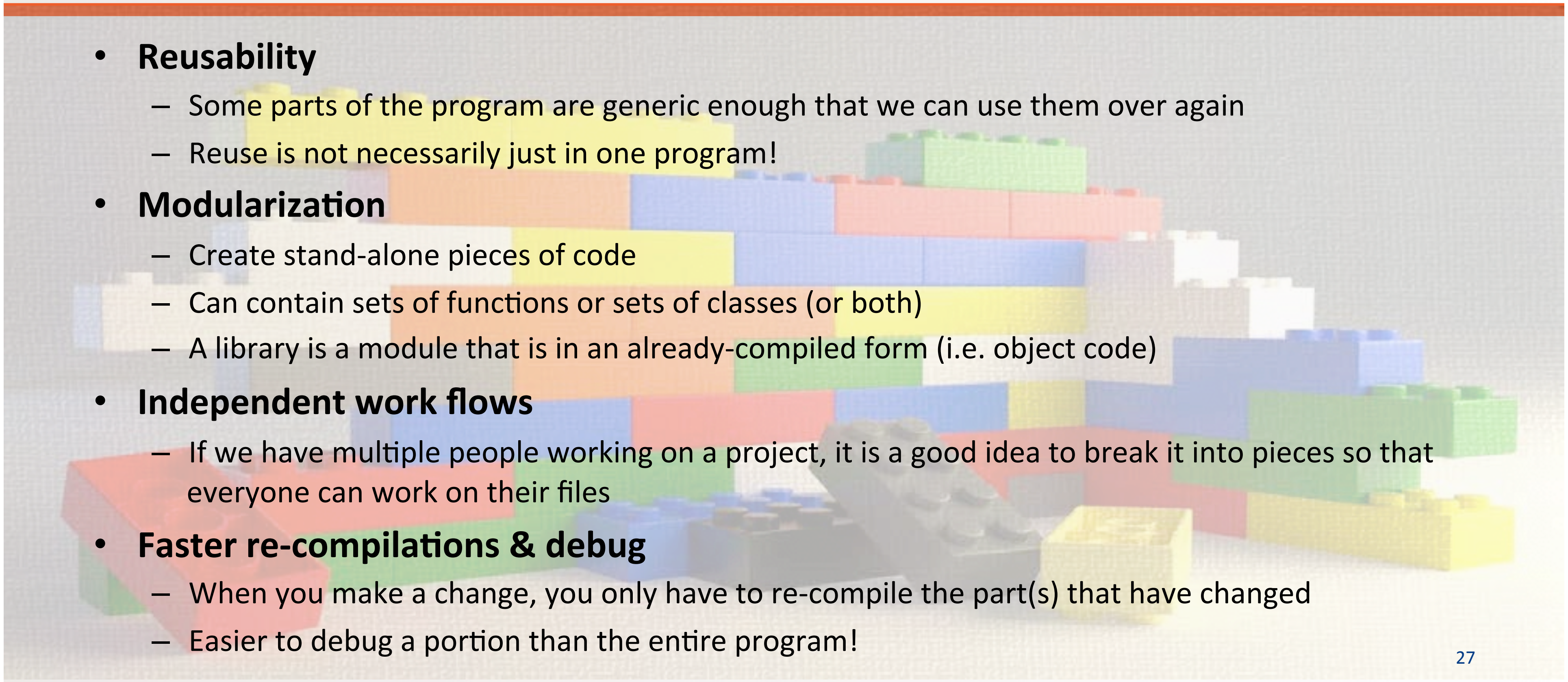
Why?

Reusability

Modularization

Independent work flows

Faster re-compilations & debug



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!

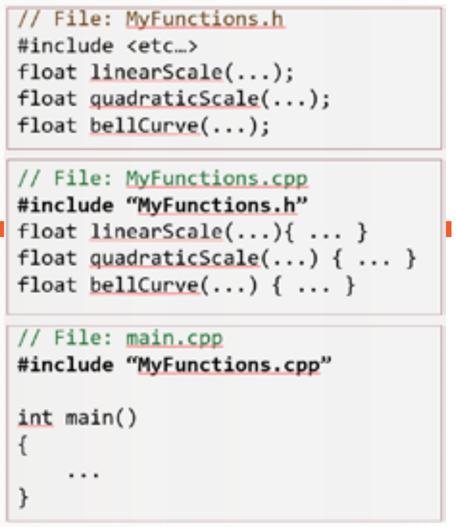
```
#include <etc…>
#include <etc…>
// function declarations
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);

// function definitions
float linearScale(...){ ... }
float quadraticScale(...) { ... }
float bellCurve(...) { ... }

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

```
// File: MyFunctions.h
#include <etc…>
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);
```

```
// File: MyFunctions.cpp
#include “MyFunctions.h”
float linearScale(...){ ... }
float quadraticScale(...) { ... }
float bellCurve(...) { ... }
```

```
// File: main.cpp
#include “MyFunctions.cpp”

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

Compiling Everything…



```
// File: MyFunctions.h
#include <etc…>
float linearScale(...);
float quadraticScale(...);
float bellCurve(...);
```

```
// File: MyFunctions.cpp
#include "MyFunctions.h"
float linearScale(...){ ... }
float quadraticScale(...) { ... }
float bellCurve(...) { ... }
```

```
// File: main.cpp
#include "MyFunctions.cpp"

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

```
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!!…

Imagine a situation with a lot more files and sub-files…

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

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** are kept in a file called ***Makefile***.

- Makefile is code written for and in Linux OS code

Makefile

- The file must be called “makefile” (or “Makefile”)
- Put all the instructions you’re going to use in there
  - There is a syntax to follow for makefiles
  - Just type “make” at the prompt, instead of all the g++ commands
  - There is an online manual for “make”:
    From a Linux prompt, type: “`man  make`”
- Makefiles can easily be used to do other OS-related stuff
  - Like “clean up” your area, for example, by removing files

Syntax of a Make

```
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
```

Target "all" programs in this project

dependencies

recipe lines

TAB spaces!

Target something specific (give it a short-cut name, like ProgY)

Doesn't have to be compiling instructions only!

Syntax of a Make

```
all: Exercise1 Exercise2

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

#This next one’s a doozy
Exercise2: ex2.cpp
	g++ -std=c++11 –Wall ex2.cpp –o ex2
```

Target "all" programs in this project

Dependencies that are declared below

# is for commenting

Note: These are TAB characters in there!

There Are Several Ways To Do This Piece-wise Approach

- See “example1” and “example2” in the demo code for this lecture (**demo_lecture14**)
- **example1**: similar to the one we just went through
- **example2**: by re-arranging headers, we can make one compile command (simpler, but also more limiting)

YOUR TO-DOs

- ❑ HW 8 due **Tue**. 11/28 in CLASS!
- ❑ Lab 8 due **Wed**. 11/29 at NOON!

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

HAVE A HAPPY THANKSGIVING!!!

</LECTURE>