# Compiling Separate C++ Files
# Strings & Vectors

**CS 16: Solving Problems with Computers I**
**Lecture #14**

Ziad Matni
Dept. of Computer Science, UCSB

# Announcements

- Lab #8 is due on Monday, 11/21 at 8 AM
- Homework #13 is due on Thursday, 11/17

# Lecture Outline

- Compiling Separate Files in C++

*CH. 8*

- Strings
  - **c-string** vs. **string** Class in C++

- Introduction to Vectors

# 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.h”
#include <iostream>, etc…

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

# Compiling Everything…

```
g++ -c MyFunctions.cpp
```

*(creates MyFunctions.o)*

```
g++ -c main.cpp
```

*(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

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

clean:
        rm *.o ProgX
```

*target*

*dependencies*

*recipe lines*

# What is a String?

- Characters connected together in a sequence

| P | i | k | a | c | h | u |
|---|---|---|---|---|---|---|

| H | i |  | M | o | m | ! |
|---|---|---|---|---|---|---|

# C strings vs. C++ strings

- Strings in C++ and Strings in C
  - C++ is meant to be backwards compatible with C
  - C has one way of dealing with strings,
    while C++ has another
- C++'s use is much easier and safer with memory allocation
  - This is what you've learned so far with <string>
  - Let's briefly review the other (older) way with C-strings…

# What's a C++ Programmer to Do?!

- Be aware of 3 types of variables that deal with a bunch of connected characters…

- An "ordinary" array of characters
  - Like any other array:
    no special properties that other arrays do not have

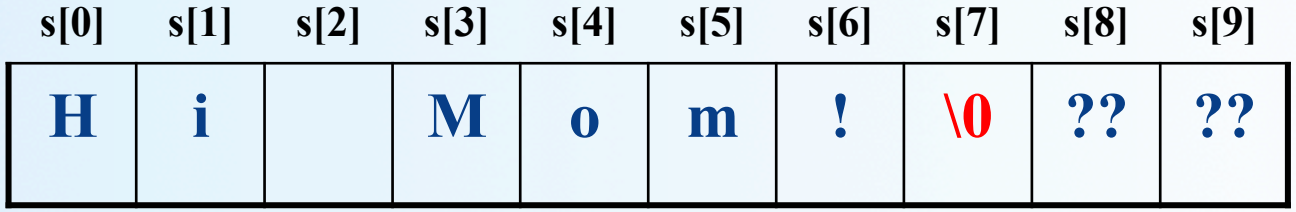
- A C-string
  - An array of characters terminated by the **null character '\0'**
  - The null character has an ASCII code of **0**.
  - Library for dealing with these types: **<cstring>**

- A C++ string object
  - An instance of a "class" data type – used a "black box"
  - Library for dealing with these types: **<string>**

# The C-String

- An array of characters that terminates in the null character
  - This terminates the actual string, but not the array necessarily

- Example : a C-string stores “Hi Mom!” in a character array of size 10
  - The characters of the word “Hi Mom!” will be in positions with indices 0 to 6
  - There will be a null character at index 7, and the locations with indices 8 to 9 will contain some unknown value.
  - But we don’t care about positions 8 and 9!
  - It's the null character at index 5 that makes this otherwise ordinary character array a C-string.



# C-string Declaration

- To declare a C-string variable, use the syntax:

```
char Array_name[ Maximum_C_String_Size + 1 ];
```

  - The "+ 1" reserves the additional character needed by '\0'

- The library to include is <cstring>
  - This is the same library as <string.h>
  - Note: This is **NOT** the same library as <string>

# Initializing a C-string

- To initialize a C-string during declaration:

```
char my_message[20] = "Hi there.";
```

  - Don't add the null character: it is added for you

- Another alternative:

```
char short_string[ ] = "abc";
```

- But you cannot do this with C-Strings:

```
char short_string[ ] = {'a', 'b', 'c'};
```

  - Because **'\0'** will not be inserted in the array…

# Don't Change '\0'
## *or How to Best Process C-Strings*

- Do not to replace the null character when manipulating indexed variables in a C-string
  - If the null character is lost, the array cannot act like a C-string
    - Commonly used process:

```
int index = 0;
while ( (our_string[index] != '\0')
     && (index < SIZE) ) {

        our_string[index] = 'X';
         index++;
    }
```

  - Note that this code depends on finding the null character and not exceeding the size of the C-string

# Assignment in C-Strings

- You cannot do the following with C-Strings:

```
a_string = "Hello";
```

  - The assignment operator does not work with C-strings
    (but it does with the C++ string class, as I'm sure you know…)

- With C-Strings, you have to use the **strncpy** function
  - Found in the <cstring> library

```
#include <cstring>
        …
    char a_string[10] = "xxxxxxxxx"; // There are 9 x
    char word[7] = "Hello!";
    strncpy (a_string, word, 7);
    cout << a_string;
    // Note that now, a_string[6] = '\0'
    // So this will print out a_string as: "Hello!"
    //                                NOT as: "Hello!\0xx"
```

# Comparisons in C-strings

- Nor does the == operator work as expected with C-strings
  - Instead, use function **strcmp**

- Example:

```
#include <cstring>
    …
    if (strcmp(c_string1, c_string2))
        cout << "Strings are not the same.";
    else
        cout << "String are the same.";
```

Note the negative logic:
**strcmp** returns 0 for *false*
and a non-0 for *true*

# The Standard C++ **string** Class

- The strings we know and love…
- The string class allows the programmer to treat strings as a basic data type
  - No need to deal with the implementations of C-strings
- The string class is defined in the <string> library and the names are in the standard namespace
- We've already discussed many different member functions that are extremely useful to use
  - .length(), .erase(), .substr(), .find(), etc…
  - Display 8.7 in your textbook lists some of these

# Recall the Differences in String Inputs!

- What's the difference between **cin** and **getline( )** ?
  - **cin** stops at each whitespace
    - Whitespace includes: space, newline, tab characters
  - **getline( )** gets the entire line up until a certain character
    - Default end character is newline (\n).

# Another Version of **getline**

- The versions of **getline** we have seen, stop reading at the end of a line marker **'\n'**
  - That's the default setting

- **getline** can also stop reading at a character specified in the argument list
  - This code, for example,
    stops reading when a '**?**' is read

```
string line;
cout <<"Enter some input: \n";
getline(cin, line, '?');
```

# **getline** Declarations

- These are the declarations of the versions of getline for string objects we have seen

```
istream& getline(istream& ins, string& str_var, char delimiter);
istream& getline(istream& ins, string& str_var);
```

- When you use **getline** with **cin**, you only need to include **<iostream>** and **<string>**

- When you use **getline** with **ifstream**, you only need to include **<ifstream>** and **<string>**

- <iostream> includes <istream>
- <istream> is a parent library of <ifstream>

- This is a relic from older versions of C++

# cin.ignore( )

- If you want to read a certain amount of characters in an input stream all in one go (including any incident '\n'), then use the member function **ignore**.

- **ignore** takes two arguments
  - First, the maximum number of characters to discard
  - Second, the character that stops reading and discarding

- Example: `cin.ignore(1000, '\n');`

reads up to 1000 characters or to '\n'

# Copying a **C-string** to a **string**

```
char a_cstring[] = "Dude Where's My Car?";
string s;
```

- You cannot do: `s = a_cstring;`
- You cannot do: `strncpy(a_cstring, s, n);`

- You have to use the **.c_str( )** member function to convert

Example:

```
char a_cstring[] = "Dude Where's My Car?";
string s;
strcpy(a_cstring, s.c_str());
```

# Converting **string** class to numbers

- In C++11 you can convert a string class object to a number

```
int i;
double d;
string s;
i = stoi("35");  // Converts the string "35" to an integer 35
d = stod("2.5"); // Converts the string "2.5" to the double 2.5
```

- You can also convert a number to a string class object

```
string s = to_string(1.2*2);  // “2.4” stored in s
```

# Vectors

# Vectors

- Vectors are like arrays that can change size as your program runs
  - You have less to worry about with vectors re: size changes
  - But vectors consume more memory in exchange for this flexible ability to manage memory and grow automatically and dynamically in an efficient way

- Vectors, like arrays, have a base type

- To declare an empty vector with base type **int**:

```
vector<int> v;
```

  - <int> identifies vector as a *template class*
  - You can use any base type in a template class:

```
vector<double> v;
vector<string> v;
   …etc…
```

# Accessing **vector** Elements

- Vectors elements are indexed starting with 0
  - [ ]'s are used to read or change the value of an item:

```
v[i] = 42;
cout << v[i];
```

- But [ ]'s cannot be used to initialize a vector element

# Initializing **vector** Elements

- Elements are added to a vector using the member function **push_back( )**
- **push_back** adds an element in the next available position
- Example:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

# The **size** of a **vector**

- The member function **size( )** returns the number of elements in a vector

  – Example: To print each element of a vector:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);

for (int i= 0; i < sample.size( ); i++)
    cout << sample[i] << endl;
```

# The Type **unsigned int**

- The vector class member function size returns an **unsigned int** type of value
  - Unsigned int's are non-negative integers

- Some compilers will give a warning if the previous for-loop is not changed to:

```
for (unsigned int i= 0; i < sample.size( ); i++)
     cout << sample[i] << endl;
```

*However, g++ seems ok with plain old "int"…*

# Alternate **vector** Initialization

- A vector constructor exists that takes an integer argument and initializes that number of elements

- Example:

```
vector<int> v(10);
```

initializes the first 10 elements to 0

```
v.size( )
```

would then return 10

- **[ ]**'s can now be used to assign elements 0 through 9
- **push_back** is used to assign elements greater than 9

# The **vector** Library

- To use the vector class
  - Include the vector library

```
#include <vector>
```

  - Vector names are placed in the standard namespace so the usual using directive is needed:

```
using namespace std;
```

**Using a Vector**

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

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

**Sample Dialogue**

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8
```

# Defining **vector** Elements Beyond Vector Size

- Attempting to use [ ] to set a value beyond the size of a vector *may not generate an error, but it is not correct to do!*

- Example: assume integer vector **v** has 3 elements in it
  - Performing an assignment like **v[5] = 4** isn't the "correct" thing to do
  - You should **push_back( )** enough to get to element 5 first before making changes
  - **push_back** operation ensures the "correct" memory allocations are being done behind the scenes

- Even though you may not get an error, you have messed around with memory allocations and the program will probably misbehave in other ways

# **vector** Efficiency

- A vector's capacity is the number of elements allocated in memory
  - You can see what that is using the **capacity( )** member function
- **size( )** is the number of elements initialized
- When a vector runs out of space,
  *the capacity is automatically increased!*
  - A common scheme is to double the size of a vector
    - More efficient than allocating smaller chunks of memory

# Controlling **vector** Capacity

- When efficiency is an issue and you want to control memory use…
  - Member function **reserve( )** can *increase* the capacity of a vector
    - Example:

```
v.reserve(32);                 // at least 32 elements
v.reserve(v.size( ) + 10);   // at least 10 more
```

  - **resize( )** can be used to *shrink* a vector
    - Example:

```
v.resize(24);   //elements beyond 24 are lost
```

# To Dos

- Homework #13 for Thursday
- Lab #8 for Monday (8AM)

# </LECTURE>