CSCI 104
Overview
Mark Redekopp
Administrative Issues

• Preparation
  – Basic if, while, for constructs
  – Arrays, linked-lists
  – Structs, classes
  – Dynamic memory allocation and pointers
  – Recursion

• Syllabus
  – http://bits.usc.edu/cs104

• Expectations
  – I'll give you my best, you give me yours...
    • Attendance, participation, asking questions, academic integrity, take an interest
  – Treat CS104 right!
  – Let's make this fun
More Helpful Links

• Remedial modules
  – http://ee.usc.edu/~redekopp/csmodules.html

• Class website
  – http://bits.usc.edu/cs104
An Opening Example

• Consider a paper phonebook
  – Stores names of people and their phone numbers

• What operations do we perform with this data
  – You: Lookup/search
  – Phone Company: Add, Remove

• How is the data stored and ordered and why?
  – Sorted by name to make lookup faster...
  – How fast? That's for you to figure out...

• What if it was sorted by phone number or just random? What is the worst case number of records you'd have to look at to find a particular person's phone number?
Opening Example (cont.)

• Would it ever be reasonable to have the phonebook in a random or unsorted order?
  – What if the phonebook was for the residence of a town with only a few residents
  – What if there was a phonebook for Mayflies (life expectancy of 1-24 hours)
    • Might want to optimize for additions and removals
    • Plus, a mayfly doesn't have fingers to dial their phones so why would they even be trying to search the phonebook

• Main Point: The best way to organize data depends on how it will be used.
  – Frequent search
  – Frequent addition/removals
  – Addition/removal patterns (many at once or one at a time)
Why Data Structures Matter?

- Modern applications process vast amount of data
- Adding, removing, searching, and accessing are common operations
- Various data structures allow these operations to be completed with different time and storage requirements

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Insert</th>
<th>Search</th>
<th>Get-Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted List</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced Binary Search Tree</td>
<td>$O(lg\ n)$</td>
<td>$O(lg\ n)$</td>
<td>$O(lg\ n)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(lg\ n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Recall $O(n)$ indicates that the actual run-time is bounded by some expression $a*n$ for some $n > n_0$ (where $a$ and $n_0$ are constants)
# Importance of Complexity

<table>
<thead>
<tr>
<th>N</th>
<th>O(1)</th>
<th>O(log₂n)</th>
<th>O(n)</th>
<th>O(n* log₂n)</th>
<th>O(n²)</th>
<th>O(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>4.3</td>
<td>20</td>
<td>86.4</td>
<td>400</td>
<td>1,048,576</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>7.6</td>
<td>200</td>
<td>1,528.8</td>
<td>40,000</td>
<td>1.60694E+60</td>
</tr>
<tr>
<td>2000</td>
<td>1</td>
<td>11.0</td>
<td>2000</td>
<td>21,931.6</td>
<td>4,000,000</td>
<td>#NUM!</td>
</tr>
</tbody>
</table>
Abstract Data Types

• Beginning programmers tend to focus on the code and less on the data and its organization

• More seasoned programmers focus first on
  – What data they have
  – How it should be organized
  – How it will be accessed

• Consider a dictionary (word and its definition)
  – What operations do we support?
    – Lookup
    – Insert
    – Remove (may not be needed)

• This is what we call an Abstract Data Type
  – Data along with the supported operations
  – We will focus on many ADTs in this class
Transition to Object-Oriented

• Object-oriented paradigm fits nicely with idea of ADTs
  – Just as ADTs focus on data and operations performed on it so objects combine data + functions
• Objects (C++ Classes) allows for more legible, modular, maintainable code units
• Suppose you and a friend are doing an electronic dictionary app. Your friend codes the dictionary internals and you code the user-interface.
  – You don't care how they implement it just that it supports the desired operations and is fast enough
  – **Abstraction**: Separates 'what' from 'how'
  – **Encapsulation**: Shields inside from outside so that internals can be changed w/o affecting code using the object
Course Goals

• Learn about good programming practice with object-oriented design
  – Learn good style and more advanced C++ topics such as templates, inheritance, polymorphism, etc.

• Learn basic and advanced techniques for implementing data structures and analyzing their efficiency
  – May require strong fundamentals including mathematical analysis
  – This is why we couple CS 104 and CS 170
You are responsible for this on your own since it's covered in CS103

C++ LIBRARY REVIEW
(END LECTURE 1 SLIDES)
C++ Library

- String
- I/O Streams
- Vector
C++ Strings

• Why do we need the string class?
  – C style strings are character arrays (char[ ])
    • See previous discussion of why we don't like arrays
  – C style strings need a null terminator ('\0')
    “abcd” is actually a char[5] ... Why?
  – Stuff like this won't compile:
    char my_string[7] = “abc” + “def”;

• How can strings help?
  – Easier to use, less error prone
  – Has overloaded operators like +, =, [], etc.
  – Lots of built-in functionality (e.g. find, substr, etc.)
#include <iostream>
#include <string>

using namespace std;

int main() {
    string x = "hello";
    x += " world!";  
    cout << x << endl;
    cout << "The string is " << x.size() << " chars long." << endl;
    cout << "The first character is " << x.at(0) << endl;

    string y = x.substr(0,2);
    //store "he" in y
    cout << y << endl;
    cout << "The letter 'e' is at index " << y.find('e') << endl;
    return 0;
}
What is a “stream”?  
- A sequence of characters or bytes (of potentially infinite length) used for input and output.

C++ has four major libraries we will use for streams:  
- `<iostream>`
- `<fstream>`
- `<sstream>`
- `<iomanip>`

Stream models some input and/or output device  
- `fstream` => a file on the hard drive;  
- `cin` => keyboard and cout => monitor

C++ has two operators that are used with streams  
- Insertion Operator “<<”  
- Extraction Operator “>>”
C++ I/O Manipulators

• The `<iomanip>` header file has a number of “manipulators” to modify how I/O behaves
  – Alignment: internal, left, right, setw, setfill
  – Numeric: setprecision, fixed, scientific, showpoint
  – Other: endl, ends, flush, etc.

• Use these inline with your `cout/cerr/cin` statements
  – double pi = 3.1415;
  – cout << setprecision(2) << fixed << pi << endl;
C++ Console Input

• cin can be used to accept data from the user
  – int x;
  – cout << "Enter a number: ";
  – cin >> x;

• What if the user does not enter a valid number?
  – Check cin.fail( ) to see if the read worked
  – Use cin.clear( ) & cin.ignore(...) on failure

• What if the user enters multiple values?
  – cin reads up until the first piece of whitespace
  – cin.getline() can read a max number of chars until it hits a delimeter (special character defaulting to \n but can be passed as an argument)
    cin.getline(buf, 80, ';') // reads everything through a ';'
    // stopping after 80 chars if no ';
  – The <string> header defines a getline(...) method that will read an entire line (including whitespace):
    string x;
    getline(cin,x,';'); // reads everything through a ';'
Understanding Extraction

User enters value “512” at 1\textsuperscript{st} prompt, enters “123” at 2\textsuperscript{nd} prompt

```cpp
int x=0;

cout << “Enter X: “;

cin >> x;

int y = 0;

cout << “Enter Y: “;

cin >> y;
```

- \textbf{X} = 0, \textbf{cin} =
- \textbf{X} = 512, \textbf{cin}.fail() is false
- \textbf{Y} = 0, \textbf{cin} =
- \textbf{Y} = 123, \textbf{cin}.fail() is false
Understanding Extraction

- User enters value “23 99” at 1st prompt, 2nd prompt skipped

```cpp
int x=0;
cout << "Enter X: ";
cin >> x;
```

```
X = 0 cin = 2 3 9 9 
```

```
cin.fail() is false
```

```cpp
int y = 0;
cin >> x;
```

```
X = 23 cin = 9 9 
```

```
cin.fail() is false
```

```cpp
int y = 0;
cout << "Enter Y: ";
cin >> y;
```

```
Y = 0 cin = 9 9 
```

```
cin.fail() is false
```

```
Y = 99 cin = 
```

int x = 0;

cout << "Enter X: ";

int y = 0;

cout << "Enter Y: ";

cin >> x;

cin >> y;

cin.fail() is false

cin.fail() is true

User enters value “23abc” at 1\textsuperscript{st} prompt, 2\textsuperscript{nd} prompt fails
Understanding Extraction

User enters value “23 99” at 1st prompt, everything read as string

```cpp
string x;

cout << "Enter X: ";

gtwline(cin,x);

X = 23 99
cin = 2 3 9 9 \n EOF

cin.fail() is false

NOTE: \n character is discarded!
Understanding cin

- Things to remember
  - When a read operation on cin goes wrong, the fail flag is set
  - If the fail flag is set, all reads will automatically fail right away
  - This flag stays set until you clear it using the cin.clear() function
  - cin.good() returns true if ALL flags are false

- When you're done with a read operation on cin, you should wipe the input stream
  - Use the cin.ignore(...) method to wipe any remaining data off of cin
  - Example: cin.ignore(1000,'\n'); cin.clear();
int y = 0;

```cpp
cout << "Enter Y: ";

Y = 0

Y = 0

Y = xxx
```

```
 cin = abc \nEOF
 cin = abc \nEOF
 cin = abc \nEOF
```

```
 cin = EOF
 cin = EOF
```

```
 cin.fail() is true
```

```
 cin = EOF
```

```
 cin = EOF
```

```
 cin = EOF
```

```
 cin = EOF
```

```
 cin = EOF
```

```
 cin = EOF
```

```
 cin = EOF
```

```cpp
 cin.ignore(100, '\n');
 // doing a cin >> here will
 // still have the fail bit set
 cin.clear();
 // now safe to do cin >>
```
C++ File I/O

• Use `<fstream>` library for reading/writing files
  – Use the open( ) method to get access to a file
    ```cpp
    ofstream out; // ofstream is for writing, ifstream is for reading
    out.open("my_filename.txt") // must be a C style string!
    ```

• Write to a file exactly as you would the console!
  – `out << "This line gets written to the file" << endl;`

• Make sure to close the file when you're done
  – `out.close();`

• Use fail( ) to check if the file opened properly
  – `out.open("my_filename.txt")`
  – `if(out.fail()) cerr << "Could not open the output file!";`
When Does It Fail

- For files & string streams the stream doesn't fail until you read PAST the EOF

```cpp
char buf[40];
ifstream inf(argv[1]);
inf >> buf;
inf >> buf;
inf >> buf;
```
Which Option?

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
  vector<int> nums;
  ifstream ifile("data.txt");
  int x;
  while( !ifile.fail() ){
    ifile >> x;
    nums.push_back(x);
  }
...
}
```

Need to check for failure after you extract but before you store/use

```
#include<iostream>
#include<fstream>
using namespace std;
int main()
{
  vector<int> nums;
  ifstream ifile("data.txt");
  int x;
  while( 1 ){
    ifile >> x;
    if(ifile.fail()) break;
    nums.push_back(x);
  }
...
}
```

A stream returns itself after extraction
A stream can be used as a bool (returns true if it hasn't failed)
Validating User Input

• Reading user input is easy, validating it is hard
• What are some ways to track whether or not the user has entered valid input?
  – Use the fail( ) function on cin and re-prompt the user for input
  – Use a stringstream for data conversions and check the fail( ) method on the stringstream
  – Read data in as a string and use the cctype header to validate each character (http://www.cplusplus.com/reference/clibrary/cctype/)
  – for(int i=0; i < str.size(); i++)
    if( ! isdigit(str[i]) )
      cerr << “str is not a number!” << endl
C++ String Stream

• If streams are just sequences of characters, aren't strings themselves like a stream?
  – The <sstream> library lets you treat C++ string objects like they were streams

• Why would you want to treat a string as a stream?
  – Buffer up output for later display
  – Parse out the pieces of a string
  – Data type conversions
    • This is where you'll use stringstream the most!

• Very useful in conjunction with string's getline(...)
C++ String Stream

- Convert numbers into strings (i.e. 12345 => "12345")

```cpp
#include <sstream>
using namespace std;

int main()
{
    stringstream ss;
    int number = 12345;
    ss << number;

    string strNumber;
    ss >> strNumber;

    return 0;
}
```

sstream_test1.cpp
C++ String Stream

• Convert string into numbers [same as atoi()]

```cpp
#include<stringstream>
using namespace std;

int main()
{
    stringstream ss;
    string numStr = "12345";
    ss << numStr;

    int num;
    ss >> num;
    return 0;
}
```

sstream_test2.cpp
C++ String Stream

• Beware of re-using the same stringstream object for multiple conversions. It can be weird.
  – Make sure you clear it out between uses and re-init with an empty string
• Or just make a new stringstream each time

```cpp
stringstream ss;

//do something with ss

ss.clear();
ss.str(" ");
// now you can reuse ss
// or just declare another stream
stringstream ss2;
```
Choices

Where is my data?

- Keyboard
  (use _____)
- File
  (use _____)
- String
  (use ______)

Do I know how many?

- Yes
- No
Choices

Is it delimited?

Yes

No

What type of data?

Text

Integers/Doubles
getline() and stringstreams

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use >>?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '
' look the same to >> and it will skip over them)
- We can use getline() to get the whole line, then a stringstream with >> to parse out the pieces

```cpp
int num_lines = 0;
int total_words = 0;

ifstream myfile(argv[1]);

string myline;
while( getline(myfile, myline) ){
    stringstream ss(myline);
    string word;
    while( ss >> word )
    {
        total_words++;
        num_lines++;
    }
}

double avg = 
    (double) total_words / num_lines;

cout << "Avg. words per line: ";
cout << avg << endl;
```

The fox jumped over the log.
The bear ate some honey.
The CS student solved a hard problem.
Using Delimiters

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line.
- Can we use `>>`?
  - No, it doesn't differentiate between different whitespace (i.e. a ' ' and a '
' look the same to `>>` and it will skip over them).
- We can use `getline()` to get the whole line, then a `stringstream` with `>>` to parse out the pieces.

```cpp
vector<string> mywords;
ifstream myfile(argv[1]);
string myline;
getline(myfile, myline, '(');
// gets "garbage stuff " // and throws away '('
getline(myfile, myline, ')');
// gets "words I care about" // and throws away ')'
stringstream ss(myline);
string word;
while( ss >> word ) {
    mywords.push_back(word);
}
```

**Text file:**
```
garbage stuff (words I care about) junk
```

**Output:**
```
0 1 2 3
```

- `"words"`  
- `"I"`  
- `"care"`  
- `"about"`
Choosing an I/O Strategy

• Is my data delimited by particular characters?
  – Yes, stop on newlines: Use getline()
  – Yes, stop on other character: User getline() with optional 3rd character
  – No, Use >> to skip all whitespaces and convert to a different data type (int, double, etc.)

• If "yes" above, do I need to break data into smaller pieces (vs. just wanting one large string)
  – Yes, create a stringstream and extract using >>
  – No, just keep the string returned by getline()

• Is the number of items you need to read known as a constant or a variable read in earlier?
  – Yes, Use a loop and extract (>>) values placing them in array or vector
  – No, Loop while extraction doesn't fail placing them in vector

Remember: getline() always gives text/string. To convert to other types it is easiest to use >>
C++ Arrays

• What are arrays good for?
  – Keeping collections of many pieces of the same data type (e.g. I want to store 100 integers)
    – int n[100];

• Each value is called out explicitly by its index
  – Indexes start at 0:

• Read an array value:

• Write an array value
C++ Arrays

• Unfortunately C++ arrays can be tricky...
  – Arrays need a contiguous block of memory
  – Arrays are difficult/costly to resize
  – Arrays don't know their own size
  – You must pass the size around with the array
  – Arrays don't do bounds checking
  – Potential for buffer overflow security holes
  • e.g. Twilight Hack: http://wiibrew.org/wiki/Twilight_Hack
  – Arrays are not automatically initialized
  – Arrays can't be directly returned from a function
  – You have to decay them to pointers
C++ Vectors

• Why do we need the vector class?
  – Arrays are a fixed size. Resizing is a pain.
  – Arrays don't know their size (no bounds checking)
  – This compiles:
    • int stuff[5];
    • cout << stuff[-1] << " and " << stuff[100];

• How can vectors help?
  – Automatic resizing to fit data
  – Sanity checking on bounds
  – They do everything arrays can do, but more safely
    • Sometimes at the cost of performance
  – See http://www.cplusplus.com/reference/stl/
Vector Class

- Container class (what it contains is up to you via a template)
- Mimics an array where we have an indexed set of homogenous objects
- Resizes automatically

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

int main()
{
    vector<int> my_vec(5); // init. size of 5
    for(unsigned int i=0; i < 5; i++)
    {
        my_vec[i] = i+50;
    }
    my_vec.push_back(10); my_vec.push_back(8);
    my_vec[0] = 30;
    unsigned int i;
    for(i=0; i < my_vec.size(); i++)
    {
        cout « my_vec[i] « " ";
    }
    cout « endl;
    int x = my_vec.back(); // gets back val.
    x += my_vec.front(); // gets front val.
    // x is now 38;
    cout "x is " « x « endl;
    my_vec.pop_back();
    my_vec.erase(my_vec.begin() + 2);
    my_vec.insert(my_vec.begin() + 1, 43);
    return 0;
}
```
Vector Class

- **constructor**
  - Can pass an initial number of items or leave blank

- **operator[]**
  - Allows array style indexed access (e.g. myvec[1] + myvec[2])

- **push_back(T new_val)**
  - Adds a *copy* of new_val to the end of the array allocating more memory if necessary

- **size(), empty()**
  - Size returns the current number of items stored as an unsigned int
  - Empty returns True if no items in the vector

- **pop_back()**
  - Removes the item at the back of the vector (does not return it)

- **front(), back()**
  - Return item at front or back

- **erase(iterator)**
  - Removes item at specified index (use begin() + index)

- **insert(iterator, T new_val)**
  - Adds new_val at specified index (use begin() + index)

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

int main()
{
    vector<int> my_vec(5); // 5= init. size
    for(unsigned int i=0; i < 5; i++){
        my_vec[i] = i+50;
    }
    my_vec.push_back(10); my_vec.push_back(8);
    my_vec[0] = 30;
    for(int i=0; i < my_vec.size(); i++){
        cout << my_vec[i] << " ";
    }
    cout << endl;
    int x = my_vec.back(); // gets back val.
    x += my_vec.front(); // gets front val.
    // x is now 38;
    cout << "x is " << x << endl;
    my_vec.pop_back();

    my_vec.erase(my_vec.begin() + 2);
    my_vec.insert(my_vec.begin() + 1, 43);
    return 0;
}
```
Vector Suggestions

• If you don’t provide an initial size to the vector, you must add items using `push_back()`
• When iterating over the items with a for loop, used an ‘unsigned int’
• When adding an item, a copy will be made to add to the vector
• [] or `at()` return a reference to an element, not a copy of the element
• Usually pass-by-reference if an argument to avoid the wasted time of making a copy

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

int main()
{
    vector<int> my_vec;
    for(int i=0; i < 5; i++){
        // my_vec[i] = i+50; // doesn’t work
        my_vec.push_back(i+50);
    }
    for(unsigned int i=0; i < my_vec.size(); i++)
    {
        cout << my_vec[i] << " ";
    }
    cout << endl;
    my_vec[1] = 5; my_vec.at(2) = 6;
    do_something(myvec);
    return 0;
}

void do_something(vector<int> &v)
{
    // process v;
}
```