• Midterm Thursday
• Note: midterm will be out of 90, last question is bonus (10 points + additional 5 bonus)
# malloc vs. new

<table>
<thead>
<tr>
<th></th>
<th>malloc()</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>what is it</strong></td>
<td>a function</td>
<td>an operator and keyword</td>
</tr>
<tr>
<td><strong>how often used in C</strong></td>
<td>often</td>
<td>never</td>
</tr>
<tr>
<td><strong>how often used in C++</strong></td>
<td>rarely</td>
<td>often</td>
</tr>
<tr>
<td><strong>allocates memory for</strong></td>
<td>anything</td>
<td>arrays, structs, objects, primitives</td>
</tr>
<tr>
<td><strong>returns</strong></td>
<td>a (void *) (needs a cast)</td>
<td>appropriate pointer type (doesn’t need a cast)</td>
</tr>
<tr>
<td><strong>when out of memory</strong></td>
<td>returns NULL</td>
<td>throws an exception</td>
</tr>
<tr>
<td><strong>deallocating</strong></td>
<td>free</td>
<td>delete or delete[ ]</td>
</tr>
</tbody>
</table>
Overloading the “==” operator

• Remember the rules we should follow?
  ▸ here’s why; hugely subtle bug

```cpp
Foo::Foo(int val) { Init(val); }
Foo::~Foo() { delete my_ptr_; }

void Foo::Init(int val) { my_ptr_ = new int; *my_ptr_ = val; }

Foo &Foo::operator=(const Foo& rhs) {
  // bug...we forgot our "if (self == &rhs) { ... }") guard
  delete my_ptr_; 
  Init(*((rhs.my_ptr_))); // might crash here (see below)
  return *this; // always return *this from =
}

void bar() {
  Foo a(10); // default constructor
  Foo b(20); // default constructor
  a = b; // crash above; dereference delete’d pointer!!
}
```
Overloading the “=” operator

• Remember the rules we should follow?
  ‣ here’s why; hugely subtle bug

• *This is yet another reason for disabling the assignment operator, when possible!!*
Suppose that...

• You want to write a function to compare two ints:

```c
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```
Suppose that...

- You want to write a function to compare two ints, and you also want to write a function to compare two strings:

```c
// note the cool use of function overloading!

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const int &value1, const int &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
int compare(const string &value1, const string &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```
Generic Code

- The two implementations of compare are nearly identical.
  - we could write a compare for every comparable type
    - but, that’s obviously a waste; lots of redundant code!
- Instead, we’d like to write “generic code”
  - code that is **type-independent**
  - code that is **compile-time polymorphic** across types
C++: parametric polymorphism

• C++ has the notion of **templates**
  ‣ a function or class that accepts a **type** as a parameter
    • you implement the function or class once, in a type-agnostic way
    • when you invoke the function or instantiate the class, you specify (one or more) types, or values, as arguments to it
  ‣ at **compile-time**, when C++ notices you using a template
    • the compiler generates specialized code using the types you provided as parameters to the template
You want to write a function to compare two things:

```cpp
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <class T>
t int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return 0;
}
```

Function template

- Same thing, but letting the compiler infer the types:

```cpp
#include <iostream>
#include <string>

// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl;
    std::cout << compare("Hello", "World") << std::endl;  // bug!
    std::cout << compare(h, w) << std::endl;  // ok
    return 0;
}
```

`functiontemplate_infer.cc`
Function template

• You can use non-types (constant values) in a template:

```cpp
#include <iostream>
#include <string>

template <class T, int N>
void printmultiple(const T &value1) {
    for (int i = 0; i < N; ++i)
        std::cout << value1 << std::endl;
}

int main(int argc, char **argv) {
    std::string h("hello");
    printmultiple<std::string, 3>(h);
    printmultiple<const char *, 4>("hi");
    printmultiple<int, 5>(10);
    return 0;
}
```
What’s going on underneath?

• The compiler doesn’t generate any code when it sees the templated function
  ▸ it doesn’t know what code to generate yet, since it doesn’t know what type is involved

• When the compiler sees the function being used, then it understands what types are involved
  ▸ it generates the instantiation of the template and compiles it
    • the compiler generates template instantiations for each type used as a template parameter
    • kind of like macro expansion
This creates a problem...

```
 ifndef _COMPARE_H_
 define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b);

endif  // COMPARE_H_

#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20); 
    cout << endl; 
    return 0;
}
```
One solution

```cpp
#ifndef _COMPARE_H_
#define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
#endif // COMPARE_H_
```

```cpp
#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

`compare.h` `main.cc`
Another solution

```
#ifndef _COMPARE_H_
#define _COMPARE_H_

template <class T>
int comp(const T& a, const T& b);

#include "compare.cc"

#endif // COMPARE_H_
```

```
#include <iostream>
#include "compare.h"

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return 0;
}
```

```
#include <iostream>
#include "compare.cc"

template <class T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```
• Templating is useful for classes as well! Imagine we want a class that holds a pair of things
  ‣ we want to be able to:
    • set the value of the first thing, second thing
    • get the value of the first thing, second thing
    • reverse the order of the things
    • print the pair of things
#include <iostream>
#include <string>

template <class Thing> class Pair {
public:
    Pair() { }
    Thing &get_first();
    Thing &get_second();
    void set_first(Thing &copyme);
    void set_second(Thing &copyme);
    void Reverse();

private:
    Thing first_, second_;
};

#include "Pair.cc"
Pair class

template <class Thing> Thing &Pair<Thing>::get_first() {
  return first_;  
}

template <class Thing> Thing &Pair<Thing>::get_second() {
  return first_;  
}

template <class Thing> void Pair<Thing>::set_first(Thing &copyme) {
  first_ = copyme;  
}

template <class Thing> void Pair<Thing>::set_second(Thing &copyme) {
  second_ = copyme;  
}

template <class Thing> void Pair<Thing>::Reverse() {
  // makes *3* copies  
  Thing tmp = first_;  
  first_ = second_;  
  second_ = tmp;  
}
```cpp
#include <iostream>
#include <string>

#include "Pair.h"

int main(int argc, char **argv) {
    Pair<std::string> ps;
    std::string x("foo"), y("bar");

    ps.set_first(x);
    ps.set_second(y);
    ps.Reverse();
    std::cout << ps.get_first() << std::endl;

    return 0;
}
```

main.cc
C++’s standard library

- Consists of four major pieces:
  - the entire C standard library
  - C++’s input/output stream library
    - std::cin, std::cout, stringstreams, fstreams, etc.
  - C++’s standard template library (STL)
    - containers, iterators, algorithms (sort, find, etc.), numerics
  - C++’s miscellaneous library
    - strings, exceptions, memory allocation, localization
Containers!

- a container is an object that stores (in memory) a collection of other objects (elements)
  - implemented as class templates, so hugely flexible
- several different classes of container
  - sequence containers (vector, deque, list)
  - associative containers (set, map, multiset, multimap, bitset)
- differ in algorithmic cost, supported
STL: the bad

- STL containers store by value, not by reference
  - when you insert an object, the container makes a copy
  - if the container needs to rearrange objects, makes copies
    - e.g., if you sort a vector, it will make many many copies
    - e.g., if you insert into a map, that may trigger several copies
  - what if you don’t want this (disabled copy con, or copy $$)
    - you can insert a wrapper object with a pointer to the object
    - we’ll learn about these “smart pointers” later
STL vector

• A generic, dynamically resizable array
  ‣ elements are stored in contiguous memory locations
    • elements can be accessed using pointer arithmetic if you like
    • random access is \(O(1)\) time
  ‣ adding / removing from the end is cheap (constant time)
  ‣ inserting / deleting from middle / start is expensive (\(O(n)\))

Example

- see Printer.cc, Printer.h, vectorfun.cc
STL iterator

- Each container class has an associated iterator class
  - used to iterate through elements of the container
  - some container iterators support more operations than others
    - all can be incremented (\(++\) operator), copied, copy-cons’ed
    - some can be dereferenced on RHS (e.g., \(x = *it;\))
    - some can be dereferenced on LHS (e.g., \(*it = x;\))
    - some can be decremented (\(--\) operator)
    - some support random access ([ ], +, -, +=, -=, <, > operators)

http://www.cplusplus.com/reference/std/iterator/
Example

- see vectoriterator.cc
STL algorithms

• A set of functions to be used on ranges of elements
  ‣ range: any sequence that can be accessed through iterators or pointers, like arrays or some of the containers
  ‣ algorithms operate directly on values using assignment or copy constructors, rather than modifying container structure
  ‣ some do not modify elements
    • find, count, for_each, min_elements, binary_search, etc.
  ‣ some do modify elements
    • sort, transform, copy, swap, etc.

http://www.cplusplus.com/reference/algorithm/
Example

- see vectoralgos.cc
STL list

- A generic doubly-linked list
  - elements are *not* stored in contiguous memory locations
    - does not support random access (cannot do list[5])
  - some operations are much more efficient than vectors
    - constant time insertion, deletion anywhere in list
    - can iterate forward or backwards
  - has a built-in sort member function
    - no copies; manipulates list structure instead of element values
Example

• see listexample.cc
STL map

- A key/value table, implemented as a tree
  - elements stored in sorted order
    - key value must support less-than operator
  - keys must be unique
    - multimap allows duplicate keys
  - efficient lookup ($O(\log n)$) and insertion ($O(\log n)$)
Example

- see mapexample.cc
Exercise 1

• Write a C++ class that:
  ‣ is given the name of a file as a constructor argument
  ‣ has a “GetNextWord( )” method that returns the next whitespace or newline-separate word from the file as a copy of a “string” object, or an empty string once you hit EOF.
  ‣ has a destructor that cleans up anything that needs cleaning up
Exercise 2

• Take one of the books from HW2’s test_tree, and:
  ‣ read in the book, split it into words (you can use your HW2)
  ‣ for each word, insert the word into an STL map
    • the key is the word, the value is an integer
    • the value should keep track of how many times you’ve seen the word, so each time you encounter the word, increment its map element
    • thus, build a histogram of word count
  ‣ print out the histogram in order, sorted by word count
  ‣ bonus: plot the histogram on a log/log scale (use excel, gnuplot, …)
    • xaxis: log(word number), y-axis: log(word count)
Review

• What have we covered so far?

• How do you review for an exam?

• What should you be expected to know?
Intro

• System design and parameters

• Layered infrastructure

• Compilation process

• High and low level languages
Intro to C

• Multiple file programs

• C types

• Functions and arguments

• Arrays

• Pointers and pointer arithmetic
- Stacks and memory model
- Box and arrow diagrams
- Pass by value vs reference
- Static and dynamic memory allocation
- Structs and typedefs
• C preprocessor

• Header files and guards

• Namespace issues

• Internal and external variables

• Makefiles
Software and UNIX

- Unit tests
- Profilers
- OS/Language interaction
- System calls and filesystem I/O
UNIX Utilities

• grep, vi, C tools

• version control

• secure programming

• suid, overflows
C++

- differences between C/C++
- iostreams and strings
- namespaces
- classes
- references
Constructors/Templates

• copy constructors

• default constructors

• overloading operators

• templates
Next three classes

• Midterm Thursday

• Sruthi teaching next Tues/Thurs
  ‣ concurrency and event-driven programming
  ‣ subclasses and inheritance

• I’m back in town next Friday

• Good luck!