Poll Everywhere

pollev.com/cse333

Which concept did you find the most difficult in the context of Homework 2?

A. The data structures
B. C-string manipulations
C. POSIX I/O (files/directories)
D. Dynamic memory allocation
E. GDB/Valgrind
F. Style considerations
G. Prefer not to say

Systems Programming

C++ STL (end), Smart Pointers (start)

Instructor: Justin Hsia

Teaching Assistants:

Aadi Adgaonkar	Advaith Vijayakumar	Alice Fu
Andrew Vlahovich	Aviel Wood	Camden Harris
Esha Gore	Hannah Hempstead	Helena Wang
Laura Wang	Megan Frisella	Nina Tharamal
Shrihun Sankepally	Shriya Sateesh	Yusong Yan

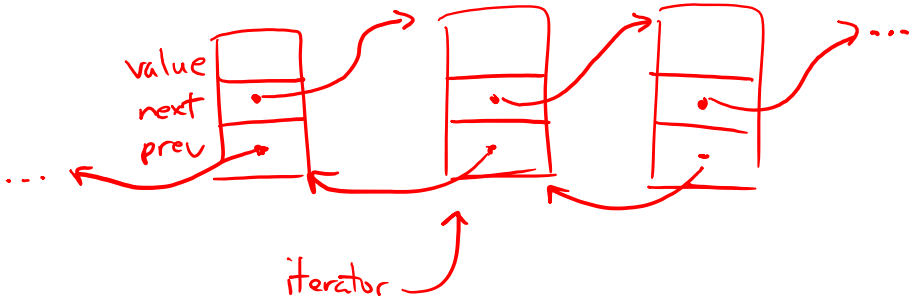
Relevant Course Information

- Exercise 8 released today and is due Monday (11/9)
  - Practice using templates and `std::map`
- Homework 3 was released last Friday, due 11/19
  - Please watch the two starter videos before section tomorrow!
- Mid-quarter survey (Canvas) due Friday (11/6)
- We are currently grading a lot (*e.g.*, HW2, Midterm, Ex7); please have patience while we work through everything
  - For exercises, in particular, reference sample solutions in the meantime

Lecture Outline (1/2)

- **STL (finish)**
  - **List**
  - **Map**
- Smart Pointers Intro

STL `list`



- A generic doubly-linked list
  - https://cplusplus.com/reference/list/list/
  - Elements are ***not*** stored in contiguous memory locations
    - Does not support random access (*e.g.*, 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 copies pointers instead of list elements
    - Doesn't copy! Manipulates list structure instead of element values

`list` Example

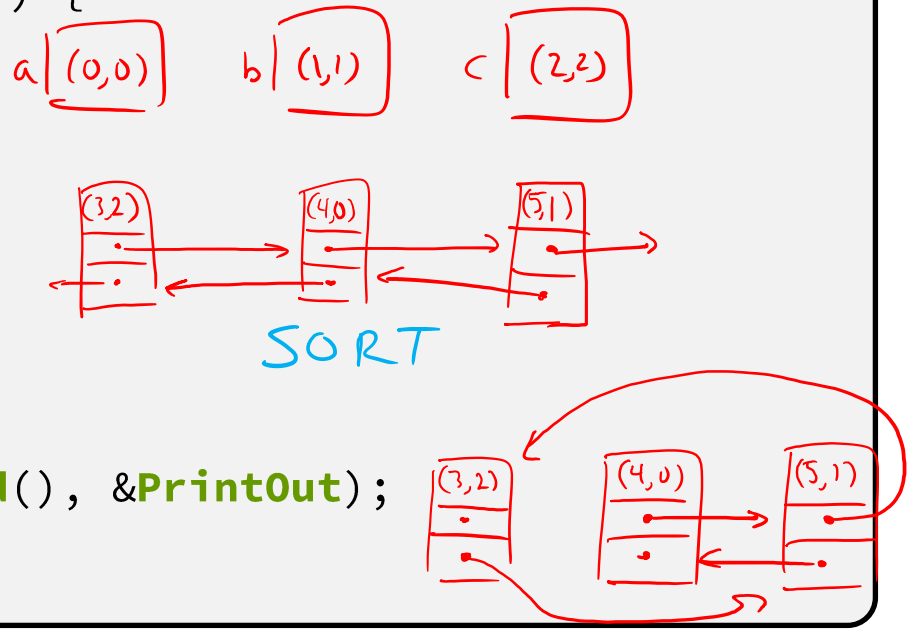
```
#include <algorithm>
#include <list>
#include "Tracer.h"
using std::…;

void PrintOut(const Tracer& p) {
  cout << " printout: " << p << endl;
}

int main(int argc, char* argv[]) {
  Tracer a, b, c;
  list<Tracer> lst;

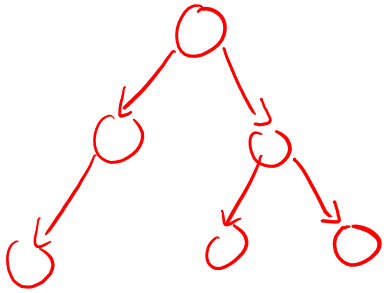
  lst.push_back(c);
  lst.push_back(a);
  lst.push_back(b);
  cout << "sort:" << endl;
  lst.sort();
  cout << "done sort!" << endl;
  for_each(lst.begin(), lst.end(), &PrintOut);
  return EXIT_SUCCESS;
}
```

listexample.cc



STL `map`

- One of C++'s *associative* containers: a key/value table, implemented as a search tree
  - https://cplusplus.com/reference/map/map/
  - General form: `map<key_type, value_type> name;`



independent types

- Properties:
  - Keys must be *unique* (use `multimap` instead for duplicate keys)
  - Efficient lookup ($\mathcal{O}(\log n)$) and insertion ($\mathcal{O}(\log n)$)
    - Access `value` via `name[key]`
  - Elements are type `pair<key_type, value_type>` and are stored in *sorted* order (key is field `first`, value is field `second`)
    - Key type must support less-than operator (`<`)

map Example

#include <map>

type of element in map<Tracer,Tracer>

```
void PrintOut(const pair<Tracer, Tracer>& p) {
  cout << "printout: [" << p.first << "," << p.second << "]" << endl;
}

int main(int argc, char* argv[]) {
  Tracer a, b, c, d, e, f;
  map<Tracer, Tracer> table;
  map<Tracer, Tracer>::iterator it;

  table.insert(pair<Tracer, Tracer>(a, b));
  table[c] = d;
  table[e] = f;
  cout << "table[e]:" << table[e] << endl;
  it = table.find(c);   // returns iterator (end if not found)
    // should check if found here before accessing element
  cout << "PrintOut(*it), where it = table.find(c)" << endl;
  PrintOut(*it);

  cout << "iterating:" << endl;
  for_each(table.begin(), table.end(), &PrintOut);

  return EXIT_SUCCESS;
}
```

} equivalent behavior

mapexample.cc

Basic `map` Usage

- `animals.cc`

Basic `map` Usage

- `animals.cc`
  - https://www.youtube.com/watch?v=jofNR_WkoCE



Homegrown `pair<>`

Usage we've seen:

```
pair<std::string, std::string> p;
p.first
p.second
```

```
template <typename T1, typename T2> struct Pair {
    // methods here – ctor, cctor, op=, dtor as needed

    T1 first;
    T2 second;
};
```

Note: just a bag of data, so struct works instead of class
↳ automatically makes first & second public

Unordered Containers (C++11)

- `unordered_map`, `unordered_set`
  - And related classes `unordered_multimap`, `unordered_multiset`
  - Average case for key access is $\mathcal{O}(1)$
    - But range iterators can be less efficient than ordered `map`/`set`
  - See *C++ Primer*, online references for details

Lecture Outline (2/2)

- STL (finish)
  - List
  - Map
- **Smart Pointers Intro**

Motivation

- We noticed that STL was doing an enormous amount of copying
- A solution: store pointers in containers instead of objects
  - But who's responsible for deleting and when???

C++ Smart Pointers

- A smart pointer is an *object* that stores a pointer to a heap-allocated object
  - A smart pointer looks and behaves like a regular C++ pointer
    - By overloading `*`, `->`, `[]`, etc.
  - These can help you manage memory
    - The smart pointer will delete the pointed-to object *at the right time* including invoking the object's destructor
      - When that is depends on what kind of smart pointer you use
    - With correct use of smart pointers, you no longer have to remember when to `delete` new'd memory!

A Toy Smart Pointer

- We can implement a simple one with:
  - A constructor that accepts a pointer
  - A destructor that deletes the pointer
  - Overloaded `*` and `->` operators that access the pointer

ToyPtr Class Template

ToyPtr.h

```
#ifndef TOYPTR_H_
#define TOYPTR_H_

template <typename T> class ToyPtr {
 public:
  ToyPtr(T* ptr) : ptr_(ptr) { }    // constructor
  ~ToyPtr() { delete ptr_; }        // destructor // clean up

  T& operator*() { return *ptr_; }  // * operator
  T* operator->() { return ptr_; }  // -> operator

 private:
  T* ptr_;   // points to something in Heap     // the pointer itself
};

#endif  // TOYPTR_H_
```

only 1 argument (this) to differentiate from multiplication

p->x ⟺ (*p).x

ToyPtr Example

usetoy.cc

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

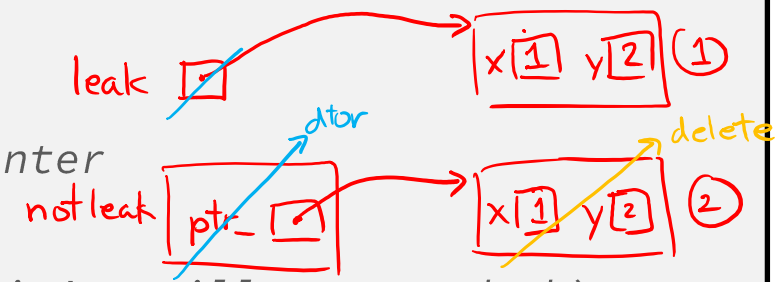
// simply struct to use
typedef struct { int x = 1, y = 2; } Point;
std::ostream& operator<<(std::ostream& out, const Point& rhs) {
  return out << "(" << rhs.x << "," << rhs.y << ")";
}

int main(int argc, char* argv[]) {
  // Create a raw ("not smart") pointer
  Point* leak = new Point;

  // Create a "smart" pointer (OK, it's still pretty dumb)
  ToyPtr<Point> notleak(new Point);

  std::cout << "     *leak: " << *leak << std::endl;      //(1,2)
  std::cout << "   leak->x: " << leak->x << std::endl;    //1
  std::cout << "  *notleak: " << *notleak << std::endl;   //(1,2)
  std::cout << "notleak->x: " << notleak->x << std::endl; //1

  return EXIT_SUCCESS;
}
```



What Makes This a Toy?

- Can't handle:
  - Arrays
  - Copying (broke the Rule of Three!)
  - Reassignment (broke the Rule of Three!)
  - Comparison
  - … plus many other subtleties…

- Luckily, others have built non-toy smart pointers for us!
  - More next lecture!

Extra Exercise #1

- 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, etc.)
    - x-axis: log(word number), y-axis: log(word count)

Extra Exercise #2

- Implement `Triple`, a class template that contains three "things," *i.e.*, it should behave like `std::pair` but hold 3 objects instead of 2
  - The "things" can be of different types

- Write a program that:
  - Instantiates several `Triple`s that contain `ToyPtr<int>`s
  - Insert the `Triple`s into a `vector`
  - Reverse the `vector`
  - Doesn't have any memory errors (use Valgrind!)
  - Note: You will need to update `ToyPtr.h` – how?