

 **Poll Everywhere**pollev.com/cse333justin

Which concept has given you the most difficulty so far in the context of Homework 2?

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

C++ STL, Smart Pointers Intro

CSE 333 Spring 2021

Instructor: Justin Hsia, Travis McGaha

Teaching Assistants:

Atharva Deodhar

Callum Walker

Cosmo Wang

Dylan Hartono

Elizabeth Haker

Kyrie Dowling

Leo Liao

Markus Schiffer

Neha Nagvekar

Nonthakit Chaiwong

Ramya Challa

Administrivia

- ❖ Exercise 7 is due Monday (5/3)
- ❖ Last day to commit & tag hw2 is Sunday (5/2)
 - Make sure these are in the repo you specified on the partner Google form
- ❖ Midterm starts Tuesday (5/4) and runs until end of Friday
 - **Topics:** everything from lecture, exercises, project, etc. up through hw2 and ex7
 - Written answers – short-answer questions and text file uploads
 - Gradescope quiz – can open, close, & submit as much as you want
 - Some discussion allowed if following the *Gilligan's Island Rule*
 - Optional exercise resubmission to earn lost points back

Lecture Outline

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

STL `list`

- ❖ A generic doubly-linked list
 - <http://www.cplusplus.com/reference/stl/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
 - Doesn't copy! Manipulates list structure instead of element values

list Example

listexample.cc

```
#include <list>
#include <algorithm>
#include "Tracer.h"
using namespace 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;
}
```

STL `map`

- ❖ One of C++'s *associative* containers: a key/value table, implemented as a search tree
 - <http://www.cplusplus.com/reference/stl/map/>
 - General form: `map<key_type, value_type> name;`
 - Keys must be *unique*
 - `multimap` allows 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

mapexample.cc

```
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);

    cout << "PrintOut(*it), where it = table.find(c)" << endl;
    PrintOut(*it);

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

    return EXIT_SUCCESS;
}
```

Basic map Usage

❖ `animals.cc`

Homegrown pair<>

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

- ❖ 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.cc

```
#ifndef TOYPTR_H_
#define TOYPTR_H_

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

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

private:
    T* ptr_;                                // the pointer itself
};

#endif // TOYPTR_H_
```

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 ("dumb") 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;
    std::cout << "    leak->x: " << leak->x << std::endl;
    std::cout << "    *notleak: " << *notleak << std::endl;
    std::cout << "    notleak->x: " << notleak->x << std::endl;

    return EXIT_SUCCESS;
}
```

What Makes This a Toy?

- ❖ Can't handle:
 - Arrays
 - Copying
 - Reassignment
 - 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: $\log(\text{word number})$, y-axis: $\log(\text{word count})$

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