



[pollev.com/cse333](https://pollev.com/cse333)

**Give a few words/adjectives to describe  
how you feel about C++ so far.**

(open-ended question)

# C++ STL, Smart Pointers Intro

## CSE 333 Winter 2023

**Instructor:** Justin Hsia

**Teaching Assistants:**

Adina Tung

Danny Agustinus

Edward Zhang

James Froelich

Lahari Nidadavolu

Mitchell Levy

Noa Ferman

Patrick Ho

Paul Han

Saket Gollapudi

Sara Deutscher

Tim Mandzyuk

Timmy Yang

Wei Wu

Yiqing Wang

Zhuochun Liu

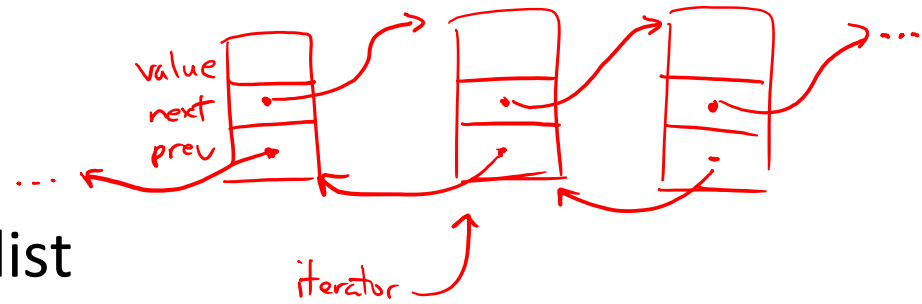
# Relevant Course Information

- ❖ Exercise 7 is due Wednesday (2/8)
- ❖ Homework 3 released today, due 2/23
  - Now in C++, but interfacing with the HW1 & HW2 C code!
- ❖ Midterm starts Thursday (2/9) and runs until end of Saturday (2/11)
  - **Topics:** everything up through hw2 and ex7
  - Gradescope quiz – can open, close, & submit as much as you want
  - Written short-answer questions
    - Look at questions *early* so you can mull them over
  - Some discussion allowed if following the *Gilligan's Island Rule*

# Lecture Outline

- ❖ **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
  - Doesn't copy! Manipulates list structure instead of element values
    - ↑ copies pointers instead of list elements

# list Example

listexample.cc

```

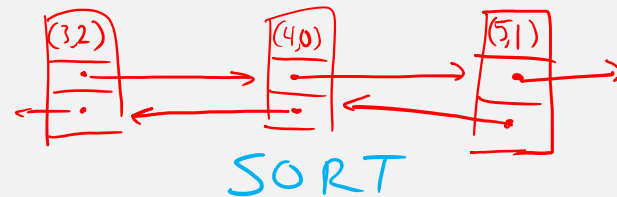
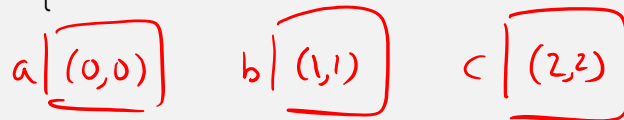
#include <list>
#include <algorithm>
#include "Tracer.h"
using namespace std;

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

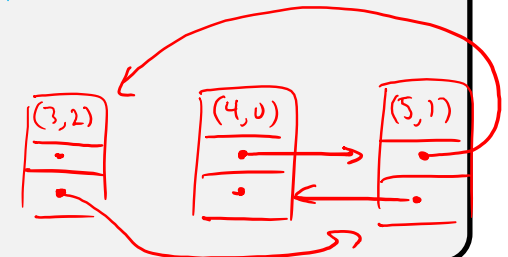
int main(int argc, char** argv) {
    Tracer a, b, c;
    list<Tracer> lst;

    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;
}

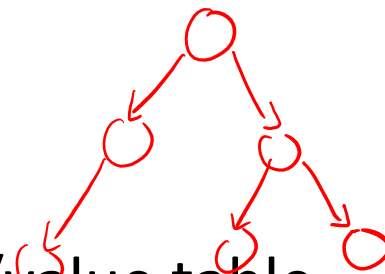
```



SORT



# 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;`
    - 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

```
#include <map>
```

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); // 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;  
}
```

type of element  
in map <Tracer, Tracer>

} equivalent behavior



# Basic map Usage

❖ `animals.cc`

# Basic map Usage

❖ `animals.cc`



- [https://www.youtube.com/watch?v=jofNR\\_WkoCE](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

- ❖ 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 // 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*  
 $p \rightarrow x \iff (*p).x$

# 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;
    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;
}

```

Diagram illustrating memory management:

- A red box labeled "leak" contains a pointer variable. A red arrow points from it to a red box labeled "x[1] y[2] (1)", representing a Point object. A red arrow labeled "leak" points from the "leak" box to the "x[1] y[2] (1)" box.
- A red box labeled "ptr\_" contains a pointer variable. A red arrow points from it to a red box labeled "x[1] y[2] (2)", representing a Point object. A red arrow labeled "notleak" points from the "ptr\_" box to the "x[1] y[2] (2)" box.
- A blue arrow labeled "dtor" points from the "ptr\_" box to the "x[1] y[2] (2)" box.
- A yellow arrow labeled "delete" points from the "ptr\_" box to the "x[1] y[2] (2)" box.

Handwritten annotations in the code:

- A red circle with the number "1" is next to `new Point` in the `leak` assignment.
- A red circle with the number "2" is next to `new Point` in the `notleak` assignment.
- Handwritten comments on the right side of the code:
  - `// (1,2)` next to `*leak`
  - `// 1` next to `leak->x`
  - `// (1,2)` next to `*notleak`
  - `// 1` next to `notleak->x`

# 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:  $\log(\text{word number})$ , 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?