C++ Smart Pointers
CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:
Dao Yi  Farrell Fileas  Lukas Joswiak
Nathan Lipiarski  Renshu Gu  Travis McGaha
Yibo Cao  Yifan Bai  Yifan Xu
About how long did Exercise 12 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I didn’t finish / I prefer not to say
Administrivia

- Exercise 12a released today
Lecture Outline

❖ STL Algorithms (review)
❖ STL Containers (continued)
❖ Smart Pointers: `std::unique_ptr`
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 most of the containers
  ▪ General form: `algorithm(begin, end, ...);`

❖ Algorithms operate directly on *range elements* rather than the containers they live in
  ▪ Make use of elements’ copy ctor, =, ==, !=, <
  ▪ Some do not modify elements
    • *e.g.* `find`, `count`, `for_each`, `min_element`, `binary_search`
  ▪ Some do modify elements
    • *e.g.* `sort`, `transform`, `copy`, `swap`
Algorithms Example

```cpp
#include <vector>
#include <algorithm>
#include "Tracer.h"

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

int main(int argc, char** argv) {
    Tracer a, b, c;
    std::vector<Tracer> vec;
    vec.push_back(c);
    vec.push_back(a);
    vec.push_back(b);

    std::cout << "sort:" << std::endl;
    std::sort(vec.begin(), vec.end());
    std::cout << "done sort!" << std::endl;

    std::for_each(vec.begin(), vec.end(), &PrintOut);
    return 0;
}
```
Lecture Outline

❖ STL Algorithms (review)
❖ STL Containers (continued)
❖ Smart Pointers: std::unique_ptr
STL list

- **Requirement**: Inserts/Deletes are O(1) time
  - Does not need to support random access (i.e. can’t do list[5])

- **Therefore**: A generic, doubly-linked list
  - Corollaries:
    - Elements are *not* stored in contiguous memory locations
    - Some operations are much more efficient than vectors, others less
    - Can iterate forward or backwards
    - Has a built-in sort member function
      - Doesn’t copy! Manipulates list structure instead of element values
list Example

```cpp
#include <list>
#include <algorithm>
#include "Tracer.h"

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

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

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

STL map

- **Requirement**: Guaranteed $O(\log n)$ lookup and insertion
  - Remember “associative” means “key -> value map”

- **Therefore**: a generic, balanced search tree
  - Elements are type `pair<key_type, value_type>` and are stored in *sorted* key order (key is field `first`, value is field `second`)
    - Access value via `name[key]`
  - Corollaries:
    - Keys must be *unique* (though `multimap` allows duplicate keys)
    - Key type must support less-than operator (`<`), value type must support default constructor
map Example

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

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

    table.insert(std::pair<Tracer, Tracer>(a, b));
    table[c] = d;
    table[e] = f;
    std::cout << "table[e]:" << table[e] << std::endl;
    it = table.find(c); // can use the invalid itr .end() to indicate "not found"
    std::cout << "PrintOut(*it), where it = table.find(c)" << std::endl;
    PrintOut(*it);
    std::cout << "iterating:" << std::endl;
    std::for_each(table.begin(), table.end(), &PrintOut);
    return EXIT_SUCCESS;
}
```
Remember This?

- **Requirement**: Guaranteed $O(\log n)$ lookup and insertion
  - Remember “associative” means “key -> value map”

- **Therefore**: a generic, balanced search tree
  - Elements are type `pair<key_type, value_type>` and are stored in sorted key order (key is field `first`, value is field `second`)
    - Access value via `name[key]`
  - Corollaries:
    - Keys must be *unique* (though `multimap` allows duplicate keys)
    - Key type must support less-than operator `<`, value type must support default constructor
Remember This?

- **Requirement**: Guaranteed $O(\log n)$ lookup and insertion
  - Remember “associative” means “key -> value map”

- **Therefore**: a generic, balanced search tree
  - Elements are type `pair<key_type, value_type>` and are stored in *sorted* key order (key is field *first*, value is field *second*)
    - Access value via `name[key]`

- Corollaries:
  - Keys must be *unique* (though `multimap` allows duplicate keys)
  - Key type must support less-than operator (<), *value type must support default constructor* other: modify the map
Unordered Containers (C++11)

- **Requirement**: Average \( O(1) \) lookup and insertion
  - `unordered_map`, `unordered_set` and related classes
    - `unordered_multimap`, `unordered_multiset`

- **Therefore**: ???
  - Hash tables can meet these requirements!
  - But range iterators can be less efficient than ordered `map`/`set`
  - See *C++ Primer*, online references for details
Lecture Outline

❖ STL Algorithms (review)
❖ STL Containers
❖ **Smart Pointers: std::unique_ptr**
Topic Goals: Smart Pointers

❖ Understand RAII semantics and various ownership options

❖ Know what `unique_ptr` provides and why it’s important it’s unique

❖ Wednesday:
  ▪ Understand various ownership models
  ▪ Compare/contrast those models
  ▪ Be able to choose the right smart pointer variant for a given task
Motivation

❖ We noticed that STL was doing an enormous amount of copying
  ▪ And if our deep-copying copy constructors were doing a lot of work, then ...

❖ A solution: store pointers in containers instead of objects
  ▪ Pointers are cheap to copy!
  ▪ But who’s responsible for deleting and when???
C++ Smart Pointers (1 of 2)

❖ 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!*
C++ Smart Pointers (2 of 2)

❖ Why does a smart pointer use the destructor?

❖ The compiler always inserts code invoking the destructor when an object goes out of scope (ie, is un-referencencable), even in cases where the programmer might forget
  ▪ Eg, multiple exit points from a function (eg, error handling)
  ▪ Eg, exiting unexpectedly due to a thrown exception (ie, exception safety!)
  ▪ Eg, deleting a dynamically-allocated item from a container
A Toy Smart Pointer

Let’s implement a simple one with:

- A constructor that accepts a pointer
- A destructor that frees the pointer
- Overloaded * and –> operators that access the pointer
ToyPtr Class Template

```
#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

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

// simple 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 dumb pointer
    int *leak = new int(5);

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

    std::cout << "*leak: " << *leak << 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!
**std::unique_ptr**

- A `unique_ptr` takes ownership of a pointer
  - Part of C++’s standard library (C++11)
  - A template: template parameter is the type that the “owned” pointer references (i.e. the `T` in pointer type `T*`)
- Defined in `<memory>`
### Using `unique_ptr`

```cpp
void Leaky() {
    int *x = new int(5); // heap-allocated
    std::cout << *x << std::endl;
} // never called delete, therefore leaked

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

void NotLeaky() {
    std::unique_ptr<Point> p(new Point); // wrapped, heap-allocated
    p->x = 5;
    std::cout << *p << std::endl;
} // never called delete, but no leak

int main(int argc, char **argv) {
    Leaky();
    NotLeaky();
    return EXIT_SUCCESS;
}
```
Transferring Ownership

❖ Use reset() and release() to transfer ownership
   ▪ release returns the pointer, sets wrapped pointer to nullptr
   ▪ reset delete’s the current pointer and stores a new one

```cpp
int main(int argc, char **argv) {
    std::unique_ptr<int> x(new int(5));
    std::cout << "x: " << x.get() << std::endl;

    std::unique_ptr<int> y(x.release());  // x abdicates ownership to y
    std::cout << "x: " << x.get() << std::endl;
    std::cout << "y: " << y.get() << std::endl;

    unique_ptr<int> z(new int(10));
    // y transfers ownership of its pointer to z.
    // z's old pointer was delete'd in the process.
    z.reset(y.release());

    return EXIT_SUCCESS;
}
```

unique3.cc
# unique_ptr Operations

```cpp
#include <memory> // for std::unique_ptr
#include <cstdlib> // for EXIT_SUCCESS

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int argc, char **argv) {
    unique_ptr<int> x(new int(5));

    int *ptr = x.get(); // Return a pointer to pointed-to object
    int val = *x;       // Return the value of pointed-to object

    // Access a field or function of a pointed-to object
    unique_ptr<IntPair> ip(new IntPair);
    ip->a = 100;

    // Deallocate current pointed-to object and store new pointer
    x.reset(new int(1));

    ptr = x.release(); // Release responsibility for freeing
    delete ptr;
    return EXIT_SUCCESS;
}
```
unique_ptrs Cannot Be Copied

- `std::unique_ptr` has disabled its copy constructor and assignment operator
  - You cannot copy a `unique_ptr`, helping maintain “uniqueness” or “ownership”

```cpp
#include <memory>  // for std::unique_ptr
#include <cstdlib> // for EXIT_SUCCESS

int main(int argc, char **argv) {
    std::unique_ptr<int> x(new int(5));  //
    std::unique_ptr<int> y(x);          //
    std::unique_ptr<int> z;             //
    z = x;                              //
    return EXIT_SUCCESS;
}
```
unique_ptr and STL

- unique_ptr\_s can be stored in STL containers
  - Wait, what? STL containers like to make lots of copies of stored objects and unique\_ptr\_s cannot be copied...

- Move semantics to the rescue!
  - When supported, STL containers will move rather than copy
    - unique\_ptr\_s support move semantics
Aside: Copy Semantics

- Assigning values typically means making a copy
  - Sometimes this is what you want
    - e.g. assigning a string to another makes a copy of its value
  - Sometimes this is wasteful
    - e.g. assigning a returned string goes through a temporary copy

```cpp
Tracer ReturnTracer(void) {
    return Tracer();  // this return might copy
}

int main(int argc, char **argv) {
    Tracer a;
    Tracer b(a);    // copy a into b
    Tracer c = ReturnTracer();  // copy return value into temp,
                                // then assign temp into c
    return EXIT_SUCCESS;
}
```
Aside: Move Semantics (C++11)

- “Move semantics” move values from one object to another without copying (“stealing”)
  - Useful for optimizing away temporary copies
  - A complex topic that uses things called “rvalue references”
    - Mostly beyond the scope of 333 this quarter

```cpp
movesemantics.cc

```std::string ReturnFoo(void) {
   std::string x("foo");
   // this return might copy
   return x;
}

int main(int argc, char **argv) {
   std::string a("hello");
   // moves a to b
   std::string b = std::move(a);
   std::cout << "a: " << a << std::endl;
   std::cout << "b: " << b << std::endl;
   // moves the returned value into b
   b = std::move(ReturnFoo());
   std::cout << "b: " << b << std::endl;
   return EXIT_SUCCESS;
}
```
Transferring Ownership via Move

- `unique_ptr` supports move semantics
  - Can “move” ownership from one `unique_ptr` to another
    - Behavior is equivalent to the “release-and-reset” combination

```cpp
int main(int argc, char **argv) {
    std::unique_ptr<int> x(new int(5));
    std::cout << "x: " << x.get() << std::endl;

    std::unique_ptr<int> y = std::move(x); // x abdicates ownership
    std::cout << "x: " << x.get() << std::endl;
    std::cout << "y: " << y.get() << std::endl;

    std::unique_ptr<int> z(new int(10));
    // y transfers ownership of its pointer to z.
    // z's old pointer was delete'd in the process.
    z = std::move(y);

    return EXIT_SUCCESS;
}
```
unique_ptr and STL Example

```cpp
int main(int argc, char **argv) {
    std::vector<std::unique_ptr<int> > vec;

    vec.push_back(std::unique_ptr<int>(new int(9)));
    vec.push_back(std::unique_ptr<int>(new int(5)));
    vec.push_back(std::unique_ptr<int>(new int(7)));

    //
    int z = *vec[1];
    std::cout << "z is: " << z << std::endl;

    //
    std::unique_ptr<int> copied = vec[1];

    //
    std::unique_ptr<int> moved = std::move(vec[1]);
    std::cout << "*moved: " << *moved << std::endl;
    std::cout << "vec[1].get(): " << vec[1].get() << std::endl;

    return EXIT_SUCCESS;
}
```