CSE 333 – SECTION
smart pointers/HW4
Memory Management With Smart Pointers

Class Bar:

Bar::Bar(int x) {
    this->x = new int(x);
}

Bar::~Bar() {
    cout << "Destructing Bar" << endl;
    delete x;
}

Class Foo:

Foo::Foo(int x) {
    shared_ptr<Bar> y(new Bar(x));
    this->x = y;
}

Foo::~Foo() {
    cout << "Destructing Foo" << endl;
}

int main() {
    shared_ptr<Foo> ptr(new Foo(5));
    ... operations on ptr ...
}

What is the expected output?

Smart pointers make memory management easier!
Useful Smart pointers

- auto_ptr (transfers ownership)
- shared_ptr (reference counting)
- weak_ptr
- scoped_ptr (no copy or assignment, manages memory only)
weak_pointer

- If an argument \( x \) is passed, and \( x \) is not empty, the `weak_ptr` object becomes part of the *owning group* of \( x \), giving access to that object's assets until *expired* without taking ownership itself (and without increasing its use_count).

"weak_ptr" is called "weak" because it points to a shared instance without incrementing the instance reference counter.
• Check whether the `weak_ptr` object is either empty or there are no more `shared_ptr` in the owner group it belongs to.

`weak_ptr::expired`
```cpp
#include <iostream>
#include <memory>

int main () {
    std::shared_ptr<int> sp(new int(10));

    std::weak_ptr<int> wp(sp);

    // wp.expired() shall be the same as (use_count()==0)
    std::cout << "1. wp " << (wp.expired() ? "is" : "is not") << " expired\n";
    sp.reset();

    std::cout << "2. wp " << (wp.expired() ? "is" : "is not") << " expired\n";
    return 0;
}
```
weak_ptr::lock

• Returns a shared_ptr with the information preserved by the weak_ptr object if it is not expired.

• If the weak_ptr object has expired (including if it is empty), the function returns an empty shared_ptr (as if default-constructed).

• Because shared_ptr objects count as an owner, this function locks the owned pointer, preventing it from being released (for at least as long as the returned object does not releases it).
weak_ptr::lock

#include <iostream>
#include <memory>

int main () {
    std::shared_ptr<int> sp1, sp2;
    std::weak_ptr<int> wp;
    // sharing group:
    // -------------
    sp1 = std::make_shared<int> (20); // sp1
    wp = sp1; // sp1, wp
    sp2 = wp.lock(); // sp1, wp, sp2
    sp1.reset(); // wp, sp2
    sp1 = wp.lock(); // sp1, wp, sp2
    std::cout << "*sp1: " << *sp1 << '\n';
    std::cout << "*sp2: " << *sp2 << '\n';
    return 0;
}
A good example of the use of a weak_ptr

• A good example would be a cache.

• 1) For recently accessed objects, you want to keep them in memory, so you hold a strong pointer to them.
• 2) Periodically, you scan the cache and decide which objects have not been accessed recently, and you get rid of the strong pointer.

• But what if that object is in use and some other code holds a strong pointer to it? If the cache gets rid of its only pointer to the object, it can never find it again. So the cache keeps a weak pointer to objects that it needs to find if they happen to stay in memory.
Using `weak_ptr` with `shared_ptr` to break cyclic references

- For smart pointers based on reference-counting, references to types can form *cycles*—that is, one object refers to a second object, the second object refers to a third object, and so on until some final object refers back to the first object.
- In a cycle, objects can't be deleted correctly (by naïve garbage collector)
- Reference-counting cannot collect cyclic (or circular) references because only one object may be collected at a time.
- To help solve this problem, we have `weak_ptr`
  - Weak references: not counted in reference counting
  - The reference cycles are avoided by using weak references for some of the references within the group.
std::shared_ptr recap

- The copy / assign operators increment a reference count
- When a shared_ptr is destroyed, the reference count is decremented
- When the reference count hits zero, the pointed-to object is deleted
cycle of shared_ptr

- #include <memory>
- using std::shared_ptr;
- class A {
  public:
  shared_ptr<A> next;
  shared_ptr<A> prev;
};
- int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;
  return 0;
}
breaking the cycle with weak_ptr

- #include <memory>
- using std::shared_ptr;
- using std::weak_ptr;
- class A {
  public:
  shared_ptr<A> next;
  weak_ptr<A> prev;
};
- int main(int argc, char **argv) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());
  head->next->prev = head;
  return 0;
}
In HW4: using weak_ptr with shared_ptr

- Can free your data structure by letting the references to the Finite State Automaton (FSA) fall out of scopes
HW 4: egrep

Build a basic subset of egrep, a pattern matching tool on a line-by-line basis.
Design your HW 4

1. Regular Expressions (RE) toFinite State Automaton (FSA)

Regular Expressions $ab^*c+d$

2. Non-deterministic execution
Design your HW 4: more details

* The FSA is basically a simply chain through states, some of which have self loops (a transition to themselves).

  FSA is a class that retains information about the states in the FSA + some useful methods

* There are three different states
  1. Start State
  2. Middle State
  3. Accept State (when you reaches the end of RE)

* There are two broad categories of edges:
  1. Epsilon edge (Free to traverse)
  2. Regular (Non-Epsilon) edge
An example struct of State (You need to use C++!)

```c
struct State {
    int c;
    State *out;
    State *out1;
    int lastlist;  // Records listid
};
```

Fragment will contains multiple states

```c
struct Fragment {
    state * starterState;  // The beginning state of a Fragment
    state **p_list;
};
```
Using switch cases in C++ for pattern match

```cpp
string::iterator it;
switch(*it) {
    case '*': {
        // Operations
    }
    case '+': {
        // Operations
    }
    case default: {
        // Operations
    }
}
```

- For each state, you need to identify whether that character only needs to appear once or multiple times.
- Need appropriate data structures to keep track of states. (Hint: Linked List and Stack).
Reminder

- The final assignment won’t have any tests be provided.
- You need to test your own code.
- Compare your output to that of `egrep`
The Rest of C++

• Smart pointers, using with STL.
  • auto_ptr (transfers ownership)
  • shared_ptr (reference counting)
  • weak_ptr (can help break cycles when jointly used with shared_ptr)
  • scoped_ptr (no copy or assignment, manages memory only)
• Subclasses, inheritance, virtual functions, dynamic vs static dispatch (function calls), vtables, constructors and destructors in subclasses
• Pure virtual functions and abstract classes
• Using class hierarchies with STL and smart pointers, assignment slicing, value vs pointer semantics
• C++ casts