CSE333 SECTION 8

Homework 4

- Any questions?

STL

- Standard Template Library
  - Has many pre-build container classes
- STL containers store by value, not by reference
- Should try to use this as much as possible

Template vs Generics

- C++ templates are similar to preprocessor macros
  - A template will be “materialized” into multiple copies with T replaced
  - Accepts primitive data types and classes
- Java generic can only be used on classes

C++ Exceptions

- Provide a way to react to exceptional in programs by transferring control to special handlers

```
try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. " << e << '\n';
  }
```

C++ Exceptions

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

C++ Exceptions

- std::exception is the base class specifically designed to declare objects to be thrown as exceptions
- Has a virtual member function called “what” that returns a null-terminated character sequence (of type char *) containing some sort of description of the exception.

```
class myexception: public exception {
  virtual const char* what() const throw() {
    return "My exception happened";
  }
} myex;
```

C++ Exceptions

C++ Standard Library also uses exceptions:

```
int main () {
  try
  {
    int* myarray= new int[1000];
  }
  catch (exception& e)
  {
    cout << "Standard exception: " << e.what() << endl;
  }
  return 0;
}
```

C++ vs Java Exceptions

- In C++, all types (including primitive and pointer) can be thrown as exception
  - only throwable objects in Java
- In C++, there is a special catch called “catch all” that can catch all kind of exceptions
  - **catch** (...) // catch all
- In Java, there is a block called finally that is always executed after the try-catch block.
  - no such block in C++
- A few other subtle differences

Exception Safety

- No-throw guarantee
- Strong exception safety: commit or rollback
- Basic exception safety: no-leak guarantee
- No exception safety: no guarantees are made

Resource Acquisition Is Initialization

- Holding a resource is tied to object lifetime:
- Resource allocation (acquisition) is done during object creation (specifically initialization), by the constructor,
- Resource deallocation (release) is done during object destruction, by the destructor.
  - If objects are destructed properly, resource leaks do not occur.

Example

```
void write_to_file (const std::string & message) {
    // mutex to protect file access
    static std::mutex mutex;

    // lock mutex before accessing file
    std::lock_guard<std::mutex> lock(mutex);

    // try to open file
    std::ofstream file("example.txt");
    if (!file.is_open())
        throw std::runtime_error("unable to open file");

    // write message to file
    file << message << std::endl;
}
```

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
  - how? by overloading *, -> , [ ] , etc.
- a smart pointer 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
  - so, if you use a smart pointer correctly, you no longer have to remember when to delete new'd memory

Smart Pointers

- The unique_ptr template is part of C++'s standard library
  - available in the new C++11 standard
- A unique_ptr takes **ownership** of a pointer
  - when the unique_ptr object is delete'd or falls out of scope, its destructor is invoked, just like any C++ object
  - this destructor invokes delete on the owned pointer

Example

```
#include <iostream> // for std::cout, std::endl
#include <memory> // for std::unique_ptr
#include <stdlib.h> // for EXIT_SUCCESS

void Leaky() {
 int *x = new int(5); // heap allocated
 (*x)++;
 std::cout << *x << std::endl;
} // never used delete, therefore leak

void NotLeaky() {
 std::unique_ptr<int> x(new int(5)); // wrapped, heap-allocated
 (*x)++;
 std::cout << *x << std::endl;
} // never used delete, but no leak
```

Why are unique_ptrs useful?

- If you have many potential exits out of a function, it's easy to forget to call delete on all of them
  - unique_ptr will delete its pointer when it falls out of scope
  - thus, a unique_ptr also helps with exception safety

unique_ptrs cannot be copied

- std::unique_ptr disallows the use of its copy constructor and assignment operator
  - therefore, you cannot copy a unique_ptr
  - this is what it means for it to be "unique"

Move

- unique_ptr supports move semantics
  - can “move” ownership from one unique_ptr to another
- old owner:
  - post-move, its wrapped pointer is set to NULL
- new owner:
  - pre-move, its wrapped pointer is delete’d
  - post-move, its wrapped pointer is the moved pointer

Example

```
int main(int argc, char **argv) {
 unique_ptr<int> x(new int(5));
 cout << "x: " << x.get() << endl;
 unique_ptr<int> y = std::move(x); // y takes ownership, x
abdicates it
 cout << "x: " << x.get() << endl;
 cout << "y: " << y.get() << endl;
 unique_ptr<int> z(new int(10));

 // z delete's its old pointer and takes ownership of y's pointer.
 // y abdicates its ownership.
 z = std::move(y);
 return EXIT_SUCCESS;
}
```

unique_ptr and STL

- unique_ptrs can be stored in STL containers!!
  - but, remember that STL containers like to make lots copies of stored objects
  - and, remember that unique_ptrs cannot be copied
  - how can this work??

- Move semantics to the rescue
  - when supported, STL containers will move rather than copy
  - luckily, unique_ptrs support move semantics

Shared Pointers

A std::shared_ptr is similar to a std::unique_ptr

- but, the copy / assign operators increment a reference count rather than transferring ownership
- after copy / assign, the two shared_ptr objects point to the same pointed-to object, and the (shared) reference count is 2
- when a shared_ptr is destroyed, the reference count is decremented
- when the reference count hits zero, the pointed-to object is deleted

Example

```
std::string foo() {
    std::string str;
    // Do cool things to or using str
    return str;
}
```

Example

```
std::string* foo() {
    std::string str;
    // Do cool things to or using str
    return &str;
}
```

Example

```
std::string* foo() {
    std::string* str = new std::string();
    // Do cool things to or using str
    return str;
}
```

Example

```
shared_ptr<std::string> foo() {
    shared_ptr<std::string> str = new std::string();
    // Do cool things to or using str
    return str;
}
```