CIS 330: UNIX and C/C++

C++ Final Details

Prof. Kevin Butler
Winter 2012
• Assignment 3 is due Thursday night
  ‣ use the mailing list for questions
• programming exercise set 6 due next Friday
  ‣ not this Friday: gives you a break and something to study for the final
• Midterms
  ‣ back Thursday (hopefully)
• Assignment 4
  ‣ out Thursday, due in a week
  ‣ will be considerably scaled back
Last time

- We learned about slicing
  - happens when a derived object is assigned to a base object
    - prevents you from mixing base / derived classes in STL containers
- Considered using pointers or wrappers to deal with this
  - pointers: lose ability to sort and must remember to delete
  - wrapper: an object that stores a pointer to some other object
    - can use copy & assign overloading so that STL does the right thing
    - but, need reference counting to know when it’s safe to delete the wrapped pointer
STL containers

- STL stores **copies of values** in containers, not pointers to object instances

  - so, what if you have a class hierarchy, and want to store mixes of object types in a single container?

    - e.g., Stock and DividendStock in the same list

    - you get sliced!

```cpp
class Stock {
    ...
};

class DivStock : public Stock {
    ...
};

main() {
    Stock s;
    DivStock ds;
    list<Stock> li;

    l.push_back(s); // OK
    l.push_back(ds); // OUCH!
}
```
STL + inheritance: use pointers?

• Store pointers to heap-allocated objects in STL containers
  ‣ Benefit: no slicing
  • Drawback: you have to remember to delete your objects before destroying the container]
  • Big drawback: sort() does the wrong thing

```cpp
#include <list>
using namespace std;

class Integer {
public:
  Integer(int x) : x_(x) { }
private:
  int x_;}

main() {
  list<Integer *> li;
  Integer *i1 = new Integer(2);
  Integer *i2 = new Integer(3);

  li.push_back(i1);
  li.push_back(i2);
  li.sort();  // waaaaaah!!}
```
An idea...

- Create a wrapper class?
  - contains a pointer to the thing we actually want to store in the STL container
    - e.g., Stock* or DividendStock*
  - overrides “<“ so sort works
  - calls delete in its destructor
  - but...STL makes many copies
    - lots of destructors are invoked
    - Causes its own problems

```cpp
#include <vector>
#include <algorithm>
using namespace std;

class Integer {
public:
  Integer(int *x) : x_(x) { }
  ~Integer() { delete x_; }
  bool operator<(const Integer &rhs) const {
    return *x_ < *(rhs.x_);
  }
private:
  int *x_;}

main() {
  vector<Integer> v;
  Integer i1(new int(2));
  Integer i2(new int(3));

  v.push_back(i1); // ok...
  v.push_back(i2); // hmm....
  // much pain...
  sort(v.begin(), v.end());
}
What we really want...

• A smarter wrapper
  ‣ contains a pointer, similar to the last slide
  ‣ overrides the copy constructor, assignment operator
    • to track # of copies of the wrapped pointer that have been made
    • a “reference count”
  ‣ has a smart destructor
    • decrements the reference count
    • calls delete if reference count falls to zero
  ‣ overrides → and * so it feels like a pointer
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
    • how? by overloading * and ->
  ‣ a smart pointer can help you manage memory
    • the smart pointer will delete the pointed-to object at the right time
      ‣ 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
C++’s auto_ptr

• The auto_ptr class is part of C++’s standard library
  ‣ it’s useful, simple, but limited
  ‣ an auto_ptr object takes ownership of a pointer
    • when the auto_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
#include <iostream> // for std::cout, std::endl
#include <memory> // for std::auto_ptr
#include <stdlib.h> // for EXIT_SUCCESS

void Leaky() {
    int *x = new int(5); // heap allocated
    (*x)++;
    std::cout << *x << std::endl;
}

void NotLeaky() {
    std::auto_ptr<int> x(new int(5)); // wrapped, heap-allocated
    (*x)++;
    std::cout << *x << std::endl;
}

int main(int argc, char **argv) {
    Leaky();
    NotLeaky();
    return EXIT_SUCCESS;
}
Why are auto_ptrs useful?

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

```cpp
int NotLeaky() {
  std::auto_ptr<int> x(new int(5));

  lots of code, including several returns
  lots of code, including a potential exception throw
  lots of code

  return 1;
}
```
auto_ptr operations

```cpp
#include <memory>    // for std::auto_ptr
#include <stdlib.h>  // for EXIT_SUCCESS

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

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

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

    // Return a reference to the value of the pointed-to object.
    int val = *x;

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

    // Reset the auto_ptr with a new heap-allocated object.
    x.reset(new int(1));

    // Release responsibility for freeing the pointed-to object.
    ptr = x.release(); // take back the pointer and manage it ourselves
    delete ptr;
    return EXIT_SUCCESS;
}
```
Transferring ownership

• The copy and assignment operators transfer ownership

  ‣ the RHS auto_ptr’s pointer is set to NULL

  ‣ the LHS auto_ptr’s pointer now owns the pointer

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

  auto_ptr<int> y(x);  // y takes ownership, x abdicates it
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  auto_ptr<int> z(new int(10));

  // z delete's its old pointer and takes ownership of y's pointer.
  // y abdicates its ownership.
  z = y;

  return EXIT_SUCCESS;
}
```
auto_ptr and STL

- Bad: auto_ptrs cannot be used with STL containers
  - a container may make copies of contained objects
    - e.g., when you sort a vector, the quicksort pivot is a copy
  - accessors will unwittingly NULL-ify the contained auto_ptr

```cpp
void foo() {
  vector<auto_ptr<int>> ivec;
  ivec.push_back(auto_ptr<int>(new int(5)));
  ivec.push_back(auto_ptr<int>(new int(6)));
  // might make copies

  auto_ptr<int> z = ivec[0]; // ivec[0] now contains a NULL auto_ptr
}
```
auto_ptr and arrays

• STL has no auto_ptr for arrays
  ‣ an auto_ptr always calls delete on its pointer, never delete[ ]
• Community supported, peer-reviewed, portable C++ libraries
  ‣ more containers, asynchronous I/O support, statistics, math, graph algorithms, image processing, regular expressions, serialization/marshalling, threading, and more
  ‣ you can download and install from:
    • http://www.boost.org/
Explicit casting in C

• C’s *explicit typecasting* syntax is simple

\[ \text{lhs} = (\text{new type}) \text{ rhs}; \]

- C’s explicit casting is used to...
  - convert between pointers of arbitrary type
  - forcibly convert a primitive type to another
    - e.g., an integer to a float, so that you can do integer division

```c
int x = 5;
int y = 2;
printf("%d\n", x / y);  // prints 2
printf("%f\n", ((float) x) / y);  // prints 2.5
```
• You can use C-style casting in C++, but C++ provides an alternative style that is more informative
  
  ‣ `static_cast<to_type>(expression)`
  ‣ `dynamic_cast<to_type>(expression)`
  ‣ `const_cast<to_type>(expression)`
  ‣ `reinterpret_cast<to_type>(expression)`
C++’s `static_cast` can convert:

- pointers to classes **of related type**
  - get a compiler error if you attempt to `static_cast` between pointers to non-related classes
  - dangerous to cast a pointer to a base class into a pointer to a derived class
- non-pointer conversion
  - float to int, etc.

`static_cast` is checked at compile time
C++’s dynamic_cast can convert:

- pointers to classes of related type
- references to classes of related type

dynamic_cast is checked at both compile time and run time:

- casts between unrelated classes fail at compile time
- casts from base to derived fail at run-time if the pointed-to object is not a full derived object
• Is used to strip or add const-ness
  ‣ dangerous!

```c
void foo(int *x) {
    *x++; 
}

void bar(const int *x) {
    foo(x);  // compiler error 
    foo(const_cast<int*>(x));  // succeeds 
}

main() {
    int x = 7;
    bar(&x);
}
```
reinterpret_cast

• casts between incompatible types
  ‣ storing a pointer in an int, or vice-versa
    • works as long as the integral type is “wide” enough
  ‣ converting between incompatible pointers
    • dangerous!
Implicit conversion

• The compiler tries to infer some kinds of conversions
  ‣ when you don’t specify an explicit cast, and types are not equal, the compiler looks for an acceptable implicit conversion

```cpp
void bar(std::string x);

void foo() {
    int x = 5.7;  // implicit conversion float -> int
    bar("hi");   // implicit conversion, (const char *) -> string
    char c = x;   // implicit conversion, int -> char
}
```
Sneaky implicit conversions

• How did the (const char *) --> string conversion work?!
  ‣ if a class has a constructor with a single parameter, the compiler will exploit it to perform implicit conversions
  ‣ at most one user-defined implicit conversion will happen
    • can do int --> Foo
    • can’t do int --> Foo --> Baz

```
class Foo {
    public:
        Foo(int x) : x_(x) { }
        int x_;  
    }

int Bar(Foo f) {
    return f.x_; 
}

int main(int argc, char **argv) {
    // The compiler uses Foo's // (int x) constructor to make // an implicit conversion from // the int 5 to a Foo.
    // equiv to return Bar(Foo(5)); // !!!
    return Bar(5); 
}
```
Avoiding sneaky implicits

- Declare one-argument constructors as “explicit” if you want to disable them from being used as an implicit conversion path
  - usually a good idea

```cpp
class Foo {
  public:
    explicit Foo(int x) : x_(x) {}
    int x_;  
};

int Bar(Foo f) {
  return f.x_;  
}

int main(int argc, char **argv) {
  // The compiler uses Foo's (int x) constructor to make an implicit conversion from the int 5 to a Foo.
  // compiler error
  return Bar(5);
}
```
Networks
Networks from 10,000ft

Internet

clients

servers

Oregon Systems Infrastructure Research and Information Security (OSIRIS) Lab
Physical layer

- Individual bits are modulated onto a wire or transmitted over radio
- Physical layer specifies how bits are encoded at a signal level e.g., a simple spec would encode “1” as +1V, “0” as -1V

1 0 1 0 1
Multiple computers on a LAN contend for the network medium

- media access control (MAC) specifies how computers cooperate
- link layer also specifies how bits are packetized and NICs are addressed
The Internet Protocol (IP) routes packets across multiple networks

- every computer has a unique Internet address (IP address)
- individual networks are connected by routers that span networks
Network layer (IP)

Protocols to:

- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination
Network layer (IP)

Packet encapsulation

- an IP packet is encapsulated as the payload of an Ethernet frame
- as IP packets traverse networks, routers pull out the IP packet from an ethernet frame and plunk it into a new one on the next network
TCP

- the “transmission control protocol”
- provides apps with reliable, ordered, congestion-controlled byte streams
- fabricates them by sending multiple IP packets, using sequence numbers to detect missing packets, and retransmitting them
- a single host (IP address) can have up to 65,535 “ports”
  - kind of like an apartment number at a postal address
TCP

- useful analogy: how would you send a book by mail via postcards?
- split the book into multiple postcards, send each one by one, including sequence numbers that indicate the assembly order
- receiver sends back postcards to acknowledge receipt and indicate which got lost in the mail
Packet encapsulation -- same as before!

Transport layer (TCP)
Applications use OS services to establish TCP streams

- the “Berkeley sockets” API -- a set of OS system calls
- clients `connect()` to a server IP address + application port number
- servers `listen()` for and `accept()` client connections
- clients, servers `read()` and `write()` data to each other
**UDP**

- the “user datagram protocol”
- provides apps with unreliable packet delivery
- UDP datagrams are fragmented into multiple IP packets
  - UDP is a really thin, simple layer on top of IP
Layer 5: session layer

- supposedly handles establishing, terminating application sessions
- RPC kind of fits in here

Layer 6: presentation layer

- supposedly maps application-specific data units into a more network-neutral representation
- encryption (SSL) kind of fits in here
Application layer

- Application protocols
  - the format and meaning of messages between application entities
  - e.g., HTTP is an application level protocol that dictates how web browsers and web servers communicate
- HTTP is implemented on top of TCP streams
Packet encapsulation -- same as before!
Packet encapsulation -- same as before!

| Ethernet header | IP header | TCP header | HTTP header | HTTP payload (e.g., HTML page) |
• Popular application-level protocols:
  ‣ **DNS**: translates a DNS name (**www.google.com**) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  ‣ **HTTP**: web protocols
  ‣ **SMTP, IMAP, POP**: mail delivery and access protocols
  ‣ **ssh**: remote login protocol
  ‣ **bittorrent**: peer-to-peer, swarming file sharing protocol

• a hierarchy of DNS servers cooperate to do this