Topics:

- Intro. to Object Oriented Pgmgs.
- Quick intro to C++ STL, including Templates

Tuesday, Aug. 29, 2005
Today’s objectives

- Give you a high-level idea of how object-oriented programming works

- Introduce you to the C++ concepts necessary to complete Lab 1
  - C++ Standard Template Library (STL)
  - Templates (very briefly!)

- Note:
  - For Lab 1, you DO NOT need to program C++ classes!!
Introduction to OOP

- Idea:
  - Object-oriented programming (e.g., using C++) is a different model of programming, as compared to functional programming (e.g., using C)

- Choosing the right model is important. Why?
  - Which model we use has a significant impact on how the problem is approached and how a solution is devised
  - That is, it can be easier to write a solution if we choose an appropriate way to model the problem.
Contrast 2 Programming Styles: (1) Procedural and (2) Object-Oriented

- Procedural programming:
  - Organize system around procedures that operate on data

- Object-oriented programming
  - Organize system around objects that receive messages
  - An object encapsulates data and operations
Example comparing 2 approaches

- **Application:** Simulate robot motion in a computer simulation (for different kinds of robots)

**Procedural (C):**

```c
typedef struct wheeled-robot {...} Wheeled;
typedef struct tracked-robot {...} Tracked;
typedef struct legged-robot {...} Legged;

void main()
{
    if (robot == wheel-type)
        move-wheeled-robot();
    else if (robot == tracked-type)
        move-tracked-robot();
    else if (robot == legged-type)
        move-legged-robot();
    ...
}
```

**Object-Oriented (C++):**

```c++
class Robot {...};

class wheeled-Robot: Robot {...
    void move-robot() {...} }
};

class tracked-Robot: Robot {...
    void move-robot() {...} }
};

class legged-Robot: Robot {...
    void move-robot() {...} }
};

void main()
{
    robot.move-robot();
    ...
}
The Object Oriented Programming Process

- Step 1:
  - Identify the data and the operations on the data
  - These begin to form the classes

- Step 2:
  - Determine how the classes interact

- (Iterate)

- Step n:
  - Create program on top of the classes
Basic Principles of Object Orientation

- Abstraction
- Encapsulation
- Modularity
- Hierarchy
- Inheritance
What is **Abstraction**?

- A model that includes most important aspects of a given problem while ignoring less important details.

An example of an item purchasing abstraction:
What is **Encapsulation**?

- Hide implementation from clients
  - Clients depend on interface
What is Modularity?

- The breaking up of something complex into manageable pieces or modules

Canteen System → Order Placement
Queue
Delivery
Billing
What is **Hierarchy**?

- Level of abstraction

Increasing abstraction

**Art**

**Music**

**Films**

**Sculptures**

Decreasing Abstraction

- R&B
- Rap
- Sci-Fi
- Action
- ...
What is Inheritance?

- Process by which one object acquires properties of another object

- Supports concept of hierarchical classification

- With inheritance, object only needs to define what makes it unique within its class

- In OOP, *Inheritance* means inheriting another object’s *interface*, and possibly its implementation
More detailed look at Object Orientation...

1. Object
2. Class
3. Attributes
4. Operation
5. Interface (Polymorphism)
6. Generalization Relationships
1. What is an **Object**?

- An object is a “smart” data structure
  - Set of state variables
  - Set of methods for manipulating state variables
- Examples – physical, conceptual, or software entities:
  - Physical entity: robot
  - Conceptual entity: chemical process
  - Software entity: linked list data structure
Objects (con’t.)

- An object advertises:
  - The types of data it will store
  - The types of operations it allows to manipulate its data (i.e., its interface)

- An object hides:
  - Its implementation of the above

- An object is something that has:
  - State
  - Behavior (i.e., operations)
  - Identity
An Object Has State

- The **state** of an object is one of the possible conditions in which an object may exist.
- The state of an object normally changes over time.
- Represented by: Attribute values + Links (relationship instances)

<table>
<thead>
<tr>
<th>Object:</th>
<th>L. Parker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>L. Parker</td>
</tr>
<tr>
<td>Employee ID:</td>
<td>9738239</td>
</tr>
<tr>
<td>Date hired:</td>
<td>Aug. 1, 2002</td>
</tr>
<tr>
<td>Status:</td>
<td>Teaching</td>
</tr>
</tbody>
</table>
An Object Has Behavior

- Behavior determines how an object acts and reacts to requests from other objects.
- Behavior is represented by the set of messages it can respond to (i.e., the operations the object can perform).

Add me to CS302 with L. Parker
(Returns: confirmation)

Registration System  \[\to\]  CS302 Course
2. What is a Class?

- A class is a description of a group of objects with common properties (attributes), behavior (operations), relationships, and semantics.
  - An object is an instance of a class.

- A class is an abstraction in that it:
  - Emphasizes relevant characteristics.
  - Suppresses other characteristics.
The Relationship Between Classes and Objects

- A class is an abstract definition of an object
  - It defines the structure and behavior of each object in the class
  - It serves as a template for creating objects

- Objects may be grouped into classes
  - A particular object of a class is an instance

Professor Parker  Professor Thomason  Professor Berry
## Sample Class

### Class

<table>
<thead>
<tr>
<th>Properties</th>
<th>Course</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>Add a student</td>
</tr>
<tr>
<td>Location</td>
<td></td>
<td>Delete a student</td>
</tr>
<tr>
<td>Days offered</td>
<td></td>
<td>Get course roster</td>
</tr>
<tr>
<td>Credit hours</td>
<td></td>
<td>Determine if it is full</td>
</tr>
<tr>
<td>Start time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>End time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Classes of Objects

- How many classes can you see?
3. What is an Attribute?

Object

Class

Attributes

CourseOffering

Attribute values

:CourseOffering

Number=CS311
startTime=1540
endTime=1655

:CourseOffering

Number=CS302
startTime=1410
endTime=1525
4. What is an Operation?

Class

Operations

CourseOffering

- addStudent
- deleteStudent
- getStartTime
- getEndTime
5. What is Polymorphism?

- The ability to hide many different implementations behind a single interface

OO Principle: Encapsulation
5. (con’t.) What is an Interface?

- A named set of operations that characterize the behavior of an element.
- The interface formalizes polymorphism

```
<<interface>>

Polygon

draw
move
scale
rotate

Triangle

Square

Pentagon
```
6. Relationships: Generalization

- A relationship among classes where one class shares the structure and/or behavior of one or more classes.
- Defines a hierarchy of abstractions in which a subclass inherits from one or more super classes.
  - Single inheritance
  - Multiple inheritance
- Generalization is an “is-a-kind of” relationship.
Example: Single Inheritance

- One class inherits from another

- **Ancestor**
  - **Superclass** (parent)
    - BankAccount
      - balance
      - name
      - number
      - Withdraw()
      - CreateStatement()
  - Generalization Relationship

- **Subclasses**
  - Checking
    - Withdraw()
  - Savings
    - GetInterest()
    - Withdraw()

- **Descendents**
Multiple Inheritance

A class can inherit from several other classes
What Gets Inherited?

- A subclass inherits its parent’s attributes and operations
- A subclass may:
  - Add additional attributes or operations
  - Redefine inherited operations (use caution)
- Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy
Example: What Gets Inherited

**Superclass**

(pARENT)

**GroundVehicle**

- weight
- licenseNumber
- register()

**Subclasses**

**Car**

- size

**Truck**

- tonnage
- getTax()
Strengths of Object Orientation

- Facilitates architectural and code reuse

- Models more closely reflect the real world
  - More accurately describe corporate data and processes
  - Are decomposed based on natural partitioning
  - Can be easier to understand and maintain

- Stability
  - A small change in requirements does not mean massive changes in the system under development
How much do you know? Let’s quiz you...

What concept does each of the following refer to?

- “Abstractions arranged in order of rank or level”
  - ANSWER: Hierarchy

- “The breaking up of something complex into manageable pieces”
  - ANSWER: Modularity

- “Extracting the essential details about an item or group of items, while ignoring the unessential details”
  - ANSWER: Abstraction

- “Enclosing all parts of an abstraction within a container”
  - ANSWER: Encapsulation
How much do you know? Let’s quiz you...

What concept does each of the following refer to?

- “Process by which one object acquires properties of another object”
  - ANSWER: Inheritance

- “The ability to hide many different implementations behind a single interface”
  - ANSWER: Polymorphism

- “An instance of a class”
  - ANSWER: An object
### History and Languages

- **1967**  Simula
- **1970-83**  Smalltalk
- **1979**  Common LISP Object System
- **1980**  Stroustrup starts on C++
- **1983**  Objective C
- **1986**  C++
- **1987**  Actor, Eiffel
- **1991**  C++ release 3.0
- **199x**  Plethora of OOP books/articles
- **1996**  Java
- **1983-89**  Language books with OOP concepts
- **1989-92**  Object-oriented design books
- **1992-present**  Object-oriented methodology books
We’ll be using C++, but there are lots of other object-oriented languages:

- Java
- Self
- Python
- Perl
- Prograph
- Modula 3
- Oberon
- Scheme
- Smalltalk Venders
- Prolog++
- Ada 95
- Object Pascal (Delphi)
- ...
C++ Standard Library

- Provides core components for I/O, strings, containers (i.e., data structures), algorithms (e.g., sort, search, merge), etc.

- C++ Standard Template Library (STL):
  - A generic library that provides solutions to managing collections of data with modern and efficient algorithms
  - All are templates → can work for arbitrary element types
  - Down-side: Can be confusing! Not necessarily self-explanatory. Have Patience!!
Quick intro to Templates

- Used to develop type-independent functions and classes

- Function template:
  - A pattern for what could become a function
  - Expanded automatically when needed

- Type of template specified as a parameter

- Compiled for each type for which it is used
Generic Function Templates

- General form of function template:

```cpp
template <class TType> return-type func-name(parameter list)
{
    // body of function
}
```

- Example:

```cpp
template <class myType> void swapargs (myType &a, myType &b)
{
    myType temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Actual type doesn’t have to be a class; can be any type (e.g., float, char, etc.). This is just a required keyword.

- These statements compile to 3 separate functions, which have 3 different types.
**Example (con’t.)**

- Previous example compiles to equivalent of:

```c
int main()
{
    int i=10, j=20;
    float x=10.1, y=23.3;
    char a='x', b='z';
    swapargs_i(i, j);  // swap integers
    swapargs_f(x, y);  // swap floats
    swapargs_c(a, b);  // swap chars
}
```

```c
void swapargs_i (int &a, int &b)
{ int temp;
    temp = a;
    a = b;
    b = temp;
}

void swapargs_f(float &a,float &b)
{ float temp;
    temp = a;
    a = b;
    b = temp;
}

void swapargs_c (char &a, char &b)
{ char temp;
    temp = a;
    a = b;
    b = temp;
}
```
Generic Class Templates

General form of class template:
```
template <class Ttype> class class-name { 
  // body of class
}
```

Example:
```
#include <iostream>
using namespace std;

template <class myType> class myClass { 
  myType x, y;
  public:
    myClass(myType a, myType b) {
      x = a;
      y = b;
    }
    myType div() { return x/y; }

  int main() {
    myClass<double> d_obj(10.0, 3.0);
    myClass<int> i_obj(10, 3);
    
    cout << "double division: " << d_obj.div() << "\n";
    cout << "integer division: " << i_obj.div() << "\n";
  }
```
Writing templates

- Beyond scope of this class!
- Can be very tedious, complex
- You just need to know how to use them
Intro to Standard Template Library

- **Main components:**
  - **Containers:**
    - Used to manage objects
    - Examples: vectors, lists, sets, stacks, queues, etc.
  - **Iterators:**
    - Used to step through elements of a collection of objects (which may be containers or subsets of containers)
    - Helpful, because they are independent of internal structure of collection of objects
  - **Algorithms:**
    - Used to process elements of collections
    - Examples: search, sort, etc.

- **Important:** all components work with arbitrary types (hence “template”)

Containers

- Two types:
  - Sequence containers:
    - **Ordered** collections in which every element has a certain position
    - Position is **dependent** on time/place of insertion
    - Position is **independent** of value of element
    - Examples: vector, deque (“deck”), list
  - Associative containers:
    - **Sorted** collections, in which the position of element depends on its value according to a sorting criterion
    - Examples: set, multiset, map, multimap
Sequence Container Example: **List**

- Implemented as a doubly-linked list
- No random access to elements
- Access to previous and next
  - \(\rightarrow\) access to arbitrary element is \(O(n)\)
- Advantage: insertion or removal is fast at any position
List code example

#include <iostream>
#include <list>
using namespace std;

int main()
{
    list<char> coll;   // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    /* print all elements; while still elements, print and remove first element
    while (! coll.empty()) {
        cout << coll.front() << ' ';    
        coll.pop_front();
    }
    
    cout << endl;
}
Strings and STL containers

- You can also use C++ string class as a type for the container class.

- Example: `list<string> t;`
  - This defines a list containing string elements

- WE’LL TALK MORE ABOUT STRINGS LATER – DON’T WORRY IF YOU AREN’T FAMILIAR WITH THEM!!
Associative Container Example: **Set**

- Associative containers typically implemented as binary tree
  - each element has 1 parent and 2 children
  - nodes to left have lesser values
  - nodes to right have greater values

- **Set** needs an iterator ... so we’ll move on to iterators
Remember main parts of STL...

- **Main components:**
  - **Containers:**
    - Used to manage objects
    - Examples: vectors, lists, sets, stacks, queues, etc.
  - **Iterators:**
    - Used to step through elements of a collection of objects (which may be containers or subsets of containers)
    - Helpful, because they are independent of internal structure of collection of objects
  - **Algorithms:**
    - Used to process elements of collections
    - Examples: search, sort, etc.
Iterators

- An object that can “iterate” (or navigate) over elements

- Fundamental operations:
  - Operator *
    - Returns the element of the current position.
  - Operator ++
    - Lets the iterator step forward to the next element (usually operator -- can be used for stepping backward)
  - Operators == and !=
    - Return whether 2 iterators represent same position
  - Operator =
    - Assigns an iterator the position of the element to which it refers
Basic member functions

- All container classes provide same basic member functions that enable them to use iterators to navigate over their elements.

- Most important functions:
  - `begin()`: returns an iterator that represents beginning of elements in container.
  - `end()`: returns an iterator that represents the end of the elements in the container (i.e., behind last element).
Two iterator types

- `container::iterator`
  - is provided to iterate over elements in read/write mode

- `container::const_iterator`
  - is provided to iterate over elements in read-only mode
Re-visit **list** example

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

int main()
{
    list<char> coll;  // list container for character elements

    // append elements from 'a' to 'z'
    for (char c='a'; c<='z'; ++c) {
        coll.push_back(c);
    }

    /* print all elements; while still elements, print and remove first element
     * while (! coll.empty()) { //replace with:
     *     cout << coll.front() << ' ';
     *     coll.pop_front();
     * }
     */

    list<char>::const_iterator pos;
    cout << endl;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
}
```
#include <iostream>
#include <set>
using namespace std;

int main()
{
    set<int> IntSet;

    IntSet coll; // set container for int values

    // Insert elements from 1 to 6 in arbitrary order. Value 1 gets inserted twice.
    coll.insert(3);
    coll.insert(1);
    coll.insert(5);
    coll.insert(4);
    coll.insert(1);
    coll.insert(6);
    coll.insert(2);

    // print elements
    IntSet::const_iterator pos;
    for (pos = coll.begin(); pos != coll.end(); ++pos) {
        cout << *pos << ' ';
    }
    cout << endl;
}
Another example (see handout)

```c
#include <stdio.h>
#include <list>
#include <string>
#include <set>
using namespace std;

#define WHITE(x) ((x < 33)||(x > 126)) // ascii whitespace?

int main(int c, char **v)
{ list<string> t; // list of transactions
  list<string>::iterator l = t.begin(); // list iterator
  char b[1<<10]; // character string buffer
  FILE *f;

  set<string> n; // set of names
  set<string>::iterator s = n.begin(); // set iterator
  int i,j,k;
  char *r;

  if ((c == 2)&&(f = fopen(v[1],"r"))) { // open file

    while (fgets(b,1<<10,f)){ // read line...
      string s(b); // make string
      l = t.insert(l,s); // insert in list
    }

    while (l != t.end()) // traverse list...
      printf("%s",(*(l++)).data()); putchar('\n'); // print transaction

  }
}
```

(Con’t next page)
Another example (con’t.)

```c
while (fgets(b,1<<10,stdin)[1]) { // input company...
    for (r = b; *r != '\n'; r++); *r = 0; // remove newline

    n.clear(); // empty set
    for (l = t.begin(); l != t.end(); l++) { // search transactions...
        for (i = j = k = 0; i < 2; i++, k += j) // move to second field
            sscanf((*l).data()+k, "%s\n", b+64, &j);

        if (strcmp(b, b+64)) continue; // company mismatch

        for (; i++ < 7; k += j) // move to name field
            sscanf((*l).data()+k, "%s\n", b+64, &j);

        while (WHITE((*l).data()[k])) k++; // skip leading whitespace
        printf("%s", (*l).data()+k); // print name
        string s((*l).data()+k); // make string
        n.insert(s); // insert in set
    }

    printf("--------------------------\n");
    for (s = n.begin(); s != n.end(); s++) { // print names...
        printf("%s", (*s).data()); putchar('\n');
    }
    return 0;
} // end if
```

printf("couldn't open %s\n", v[1]);
return 1;
```