Admin

- Midterm done!
- Assign 5 out
  - Fun, not too hefty
  - Use of outside sources
- Today's topics
  - Finish template fns, start OOP, class design/implementation
- Reading
  - Ch 8 objects/classes (today)
  - Ch 10 class templates (next)

Sort template with callback fn

```
template <typename Type>
  void Sort(Vector<Type> &v, int (cmp)(Type, Type))
  {
    for (int i = 0; i < v.size() - 1; i++) {
      int minIndex = i;
      for (int j = i+1; j < v.size(); j++) {
         if (cmp(v[j], v[minIndex]) < 0)
             minIndex = j;
      }
      Swap(v[i], v[minIndex]);
    }
  }
```

- Now can truly work for all types!
  - Client supplies function pointer to handle comparison for type

Supplying callback fn

```
int CoordCmp(coordT c1, coordT c2)
{
    if (c1.x < c2.x) return (-1);
    else if (c1.x > c2.x) return (1);
    else if (c1.y < c2.y) return (-1);
    else if (c1.y > c2.y) return (1);
    else return (0);
}

int main()
{
    Vector<coordT> pts = ... ;
    Sort(pts, CoordCmp);
```

One last convenience

- Currently, client must provide callback
  - Not as convenient when could use built-in < when it would work
- Add behavior to use < by default
- Client can supply function only when needed
  - Default argument for comparator is generic compare callback
    - OperatorCmp invokes built-in < on arguments

Final version of Sort template

```
(from CS106b cmpfn.h)
template <typename Type>
  int OperatorCmp(Type one, Type two)
  {
     if (one == two) return 0;
     return (one < two ? -1 : 1);
  }



template <typename Type>
  void Sort(Vector<Type> &v,
        int (cmp)(Type one, Type two) = OperatorCmp)
  {
     for (int i = 0; i < v.size() - 1; i++) {
         // rest of code as before
```

Use of Sort template

```
int ReverseCmp(int a, int b)
{
    if (a < b) return 1;
    else if (a > b) return -1;
    return 0;
}

int main()
{
     Vector<int> num = ...;

     Sort(num);
     Sort(num, ReverseCmp);
```

Why object-oriented programming?

- Most programs organized around data
  - Making data the focus is good fit
- Objects leverage analogy to real world
  - Time, Stack, Event, Message, etc.
- Abstraction clears away details
  - Can focus on other tasks instead
- Encapsulation provides robustness
  - Object internals can be kept private and secure
- Modularity in development
  - Design, develop, test classes independently
- Potential for reuse
  - Class is tidy package that can be re-used in other programs

Class division

Client	Interface	Implementation
Code file `client.cpp`	Header file `class.h`	Code file `class.cpp`
Contains code using objects	Contains declaration of class interface (data members and member functions)	Contains code for class member functions
`#include class.h` interface for each class used		`#include class.h` interface

Class interface in .h file

- Class interface lists data and operations
  - Data members (fields)
  - Member functions (methods)
  - Use public/private sections to control visibility to clients

```
/* File: time.h */

class Time {
  public:
     void setHour(int newValue);
     int getHour();
     void shiftBy(int dh, int dm);
     string toString();
            /* And so on... */
  private:
     int hour, minute;
};
```

Storage for objects

- A Time object has two data members
- Object about same size as comparable struct
- Declare Time object on stack

```
Time t;
```

t

hour	?
minute	?

- Each Time object has its own copy of data members
- When accessing members, it is always a particular object's copy

Accessing members

- Members accessed like struct fields
  - Usually declare on stack, but can use new for heap
  - Use . or -> depending on whether pointer or not
- Client can access *public* features

```
Time t;

t.setHour(3);
cout << t.getHour();
t.hour = 3; // only ok if field public
```

- Object being messaged is called *receiver*
- Error for client to access private member

Class implementation

- Implementation goes in .cpp file
  - Must #include class.h file
  - Contains code for member functions
  - Function name must include class scope (else assumed global function)

```
/* File: time.cpp */
#include "time.h"

void Time::setHour(int newValue)
{
    hour = newValue;
}

string Time::toString()
{
   return IntegerToString(hour) + ":"
          + IntegerToString(minute);
}
```

Implementing member functions

- Members of receiver accessible in member function

```
void Time::shiftBy (int dh, int dm) {
   hour += dh;
   minute += dm;
}
```

- Can send other messages to receiver

```
void Time::shiftBy(int dh, int dm) {
    setHour(hour + dh);
    setMinute(minute + dm);
}
```

- Special variable: this (pointer to receiver)

```
void Time::shiftBy(int dh, int dm) {
    this->hour += dh;
    this->setMinute(minute + dm);
}
```

Maintaining object consistency

- Setters can constrain to correct range

```
void Time::setHour(int newValue) {
     if (newValue < 1) hour = 1;
     else if (newValue > 12) hour = 12;
     else hour = newValue;
}

void Time::setMinute(int newValue) {
     minute = newValue % 60;
}
```

- What if data members were public?
- What is advantage of making all access, even within implementation, go through setters?

Constructors

- Special function to init newly created object
  - Data members for new object are uninitialized otherwise (not automatically set to zero as in Java)
- Called automatically when declared/allocated
  - Allocation and initialization go hand-in-hand
- Special prototype
  - Must have exact same name as class
  - No return type
  - Can have whatever parameters you need

Add Time constructor

- Declare constructor in time.h interface

```
class Time {
   public:
      Time(int hr, int min);
      void setHour(int newValue);
};
```

- Implement constructor in time.cpp

```
Time::Time(int hr, int min) {
    hour = hr;
    minute = min;
}
```

- Give args to constructor when declaring

```
Time t(2, 15);
```

Destructors

- Special function to clean up object
  - Data members may be orphaned otherwise
  - Called automatically on delete or exiting scope of object
- Special prototype
  - Same name as class prefixed with ~
  - No parameters
  - No return type
- Not always needed
  - Only if dynamically allocated members to delete, open files to close, etc.

Basic thoughts on object design

- Never let object get into malformed state
  - No public data members
  - Correctly initialize all members in constructor
  - Only provide setters if needed, be sure properly constrained
- Object is responsible for own behavior
  - Interface includes complete set of operations
  - Need to print a Time? Add print method to class, don't pull out the hour/minute fields and do it yourself
  - Same for converting time to string, comparing two times, shifting a time forward, etc.

Internal vs external representation

- Client might expect Time work in terms of hours and minutes
  - But this is difficult to manipulate internally
  - Considering mixing in AM/PM, too
  - What is required to shift time or compare?
- Consider comparing two Times:

```
bool Time::isLessThan(Time other) {
   return ((hour < other.hour) ||
    (hour == other.hour && minute < other.minute);
}
```

  - Is there a better way?

Better representation

- If Time stored in military 24-hour time?
  - Somewhat easier to shift, avoids problems with AM/PM
- If internally tracked as minutes since midnight..?
  - Trivial to implementation "lessThan" operation
  - Trivial to shift, easy to handle wrap around
- Can provide accessor for hour/minute if needed
  - Simple to compute from internal representation
- Does changing internal data affect client use?
  - What impact does making things public have on implementation flexibility?

ADTs (abstract data types)

- Client uses class as abstraction
  - Invokes public operations only
  - Internal implementation not relevant!
- Client can't and shouldn't muck with internals
  - Class data should private
- Imagine a "wall" between client and implementor
  - Wall prevents either from getting involved in other's business
  - Interface is the "chink" in the wall
    - Conduit allows controlled access between the two
- Consider Lexicon
  - Abstraction is a word list, operations to verify word/prefix
  - How does it store list? using array? vector? set? does it matter to client?