Using enums to model cards

- Consider the declaration below from card.h, simulate playing card

```cpp
class Card
{
    public:
        enum Suit {spades, hearts, diamonds, clubs};

        Card(); // default, ace of spades
        Card(int rank, Suit s);

        bool SameSuitAs(const Card& c) const;
        int GetRank() const;
        bool IsJoker() const;

    private:
        int myRank;
        Suit mySuit;
};
```
Using class-based enums

- We can't refer to Suit, we must use Card::Suit
  - The new type Suit is part of the Card class
  - Use Card::Suit to identify the type in client code
  - Can assign enum to int, but need cast going the other way

```cpp
int rank, suit;
tvector<Card> deck;
for(rank=1; rank < 52; rank++)
{
    for(suit = Card::spades; suit <= Card::clubs; suit++)
    {
        Card c(rank % 13 + 1, Card::Suit(suit));
        deck.push_back(c);
    }
}
```
Client programs and the Card class

- Do we need to know about enums to write `isFlush()`?
  - In a flush, all cards are the same suit
  - See card.h online, more functions than on previous slide
  - What about `SameSuitAs` function?
  - What about `GetRank` function?

- Think behavior before state, do we need the suit?
  - Can cards be compared for `==`? What about `<`?
  - Why would we need these operators?

- How do we write poker hand three-of-a-kind?
  - What about full house?
How do objects act like built-in types?

- We’ve used Date objects, and in many cases used the same operations that we use on ints and doubles
  - We print with operator <<
  - We add using +, +=, and ++
  - We compare using ==, <, >
- In C++ class objects can be made to act like built-in types by overloading operators
  - We can overload operator << to print to streams
  - We can overload operator == to compare Date objects
- We’ll develop a methodology that allows us to easily implement overloaded operators for classes
  - Not all classes should have overloaded operators
  - Is overloading + to be the union of sets a good idea?
Case study: the class ClockTime

- Represents hours, minutes, seconds, e.g., 1:23:47 for one hour, twenty-three minutes, 47 seconds
  - ClockTime values can be added, compared, printed

```cpp
class ClockTime
{
    public:
        ClockTime();
        ClockTime(int secs, int mins, int hours);
        int Hours() const; // returns # hours
        int Minutes() const; // returns # minutes
        int Seconds() const; // returns # seconds
}
```

- How are values represent internally (private), what are some options?
  - Do client program need to know the representation?
Using the class ClockTime

- The code below shows how the class can be used, what overloaded operators are shown?

```cpp
int h, m, s;
ClockTime total(0, 0, 0);
ClockTime max = total; // zero
while (cin >> h >> m >> s)
{
    ClockTime t(s, m, h);
    total += t;
    if (t > max)
    {
        max = t;
    }
}
cout << "total time = " << total << endl;
cout << "max time   = " << max << endl;
```
Design and Implementation Issues

- Converting to a string facilitates writing to a stream
  - We know how to write strings, conversion to a string solves many problems
  - Every class should have a `toString()` method – Java does

- An object could be in a bad state, 1 hour 72 min. 87 sec., How can this happen? How do we prevent bad state?
  - Ignore illegal values
  - Stop the program
  - Convert to something appropriate

- For ClockTime class we’ll normalize, convert to standard form
Relational operators

- Relational operators are implemented as free functions, not class member functions (Tapestry approach, not universal)
  - Needed for symmetry in some cases, see Howto E for details
  - We’ll use member function Equals to implement ==
- Print-to-stream operator "<<" must be a free function
  - We’ll use toString to implement "<<", avoid using friend functions

```cpp
ostream & operator << (ostream & os, const ClockTime & ct);
bool operator == (const ClockTime& lhs, const ClockTime& rhs);
```

- These prototypes appear in `clockt.h`, no code just prototype
  - Code in header file causes problems with multiple definitions at link time
Free functions using class methods

- We can implement `==` using the `Equals` method. Note that `operator ==` cannot access `myHours`, not a problem, why?

```cpp
bool operator == (const ClockTime& lhs, const ClockTime& rhs)
{
    return lhs.Equals(rhs);
}
```

- We can implement `operator <<` using `toString()`

```cpp
ostream & operator << (ostream & os, const ClockTime & ct)
// postcondition: inserts ct onto os, returns os
{
    os << ct.ToString();
    return os;
}
```

- Similarly, implement `+` using `+=`, what about `!=` and `<`?
Class or Data invariants

• A ClockTime object must satisfy class invariant to be valid
  ➢ Data invariant true of object as viewed by client program
  ➢ Cannot have minutes or seconds greater than 60
  ➢ What methods can break the invariant, how do we fix this?

• A private, helper function Normalize maintains the invariant

```cpp
void ClockTime::Normalize()
// post: myMinutes < 60, mySeconds < 60, represents same time
{
    myMinutes += mySeconds/60;
    mySeconds %= 60;
    myHours += myMinutes/60;
    myMinutes %= 60;
}
```
Implementing similar classes

- The class `BigInt` declared in `bigint.h` represents integers with no bound on size
  - How might values be stored in the class?
  - What functions will be easier to implement? Why?

- Implementing rational numbers like 2/4, 3/5, or –22/7
  - Similarities to `ClockTime`?
  - What private data can we use to define a rational?
  - What will be harder to implement?

- What about the `Date` class? How are its operations facilitated by conversion to absolute number of days from 1/1/1?
Niklaus Wirth

- Designed and implemented several programming languages including Pascal, Modula-2, Oberon
  
  Simple, elegant solutions are more effective, but they are harder to find than complex ones, and they require more time which we too often believe to be unaffordable

- Wrote the paper that popularized the idea of step-wise refinement
  - Iterative enhancement
  - Grow a working program

- Not a fan of C++
What’s in a file, what’s in a string?

- **Characters** make up words in English, the type `char` is used as a basic building block in C++ and other languages
  - The type `char` represents characters in different languages, encoding depends on the character set used
  - ASCII is common in C++ and other languages, limited to 128 or 256 different characters (8 bits/character)
  - Unicode is an alternative, uses 16 bits so more characters
    - Used in Java, can be used in C++, but uncommon

- **Strings** are essentially vectors/arrays of `char` values
  - Strings support catenation, find, read/write

- At a basic level, files are collections of characters
  - Especially true in Unix, other operating systems as well
Basics of the type char

- Values of type char use single quotes, not double quotes
  - 'a' as compared to "A"

- The library accessible in <cctype> (or <ctype.h>) supports character-set independent char operations

```cpp
string s = "Hello";
int k;
for(k=0; k < s.length(); k++)
{  char ch=s[k];
    if (isupper(ch))
       cout << tolower(ch) << end;
}
```

- "bool"-like functions return int values, not bool values!!
  - tolower “does the right thing” for uppercase values
Char compared to int

- A char is like a “little” int, only 8 bits as opposed to 32 (e.g.)
  - $2^8 = 256$, in practice $2^7$ values is often the limit
  - Sometimes be aware of unsigned vs signed limits on ints

- Except for printing or storing in string, a char is an int
  - Different output below?

```cpp
int k;
for(k=‘a’; k <= ‘z’; k++)
{
    cout << a << endl;
    cout << char(a) << endl;
}
```

- What about arithmetic with char types?
Char values as integers

- Char values can be compared using <, >, <=, >=, ==, !=
  - < ordering depends on character set; 'A' < 'a' in ASCII
  - Code should NOT rely on ASCII specifics, use `<cctype>` version of tolower rather than

```c
char tolower(char c)
// post: return lowercase version of c
{
    if ('A' <= c && c <= 'Z')
    {
        return c + 32;
    }
    return c;
}
```

- In practice int values are used in functions like tolower(…)

I/O, Files, chars, and trouble

- We’ve used operator >> to read from files and cin
  - Skips whitespace, can avoid this with iomanip functions
  - Sometimes want to read a line at a time or char at a time

- The function getline(..) can be used with istreams and strings
  - Reads “line” at a time --- configurable definition of line
  - Reads the ‘\n’ newline character, doesn’t store it
    - When used with operator >> this can be trouble

```cpp
string a, b;
cin >> a; getline(cin, b); // trouble!
```

- The function stream.get(ch) doesn’t skip whitespace, reads it
Using istream (istringstream) objects

- "data" file contains lines like: Joe 20 30 40 60 70

```c++
    ifstream ifile("data");
    string line,name;
    int num,count;
    double total;
    while (getline(ifile,line))
    {
        istream iline(line.c_str()); // istringstream
        iline >> name;
        total = count = 0;
        while (iline >> num) // read all numbers on line
        {
            count++;
            total += num;
        }
        cout << count << " average = " << total/count << endl;
    }
```

- The variable iline must be defined inside the outer loop, why?
Other file-reading functions

- **getline** has an optional third argument that defines when a "line" ends
  - Process data file
    - The Beatles : Let it Be
    - The Rolling Stones : Let it Bleed

    ```
    string artist, group;
    while (getline(ifile, artist, ':') &&
        getline(ifile, group))
    {
        // process artist, group
    }
    ```

- Also can read a file one char at-a-time using **input.get(ch)**
  - Doesn’t skip white space, reads every character
State machines for reading

- Sometimes the “definition” of a word changes (like the definition of a line can change with third argument to getline)
  - Using `>>` means white-space delimited words
  - What about removing comments? What about using other characters to delimit words, e.g., dashes—as this shows

- Reading is in one of several states, rules for state transitions determine how to change between states
  - In reading `//` comments there are three states: text, first-slash, comment
  - In reading `/*` comments how many states are there?
State diagram (decomment.cpp)

- Idea, read and echo characters
  - What if in comment?
- Start with simple echo program
  - Add to working program
- If we’re reading a / and find another slash, what state do we move to?
  - If not, e.g., 3/5, echo /
- If we’re reading a // comment, when does comment stop?
  - Transition from comment
Files as lines of characters

- Files are read by both computers and humans
  - Text files are composed of lines, lines composed of chars
    - Lines are simple for humans to read/process
  - Using `operator >>` to extract strings, ints, etc. doesn’t let us read files a line-at-a-time, consider file format below:

  
  Joe 20 30 40
  Sam 50 60 30 40

  - How can we read varying number of scores per line?
    - What about alternative of using a sentinel end-of-line value?

- Use `getline(..)` to read a line-at-a-time, use `istringstream (istrstream)` to process the line as a stream
State machine for /* comments */

- **Similar to // comment machine**
  - Where are characters printed/echoed?
  - Why four states?
- **State transition arcs**
  - Be sure every char covered in each state
  - In particular, slash-to-text?
  - Start to comment?
- **What about “this /* string” ?**
  - Is it hard to recognize string literals?
  - What are the issues?
Defining states

- See the program decomment.cpp for details
  - States can be identified with numbers as labels

```cpp
const int TEXT = 0;
const int FIRST_SLASH = 1;
```

- Using an enumerated type is the same idea, but gives the labels a type

```cpp
enum Suit {spades, diamonds, hearts, clubs};
```

- Can assign enum to int, but cannot assign int to enum

```cpp
Suit s = 3; // illegal
int k = spades; // legal
```