Classes: From Use to Implementation

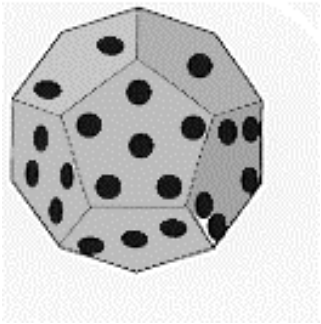
- We've used several classes, a class is a collection of objects sharing similar characteristics
  - A class is a type in C++, like `int`, `bool`, `double`
  - A class encapsulates state and behavior
  - A class is an *object factory*
- string (this is a standard class), need `#include <string>`
  - Objects: "hello", "there are no frogs", ...
  - Methods: `substr(…)`, `length(…)`, `find(…)`, `<<`

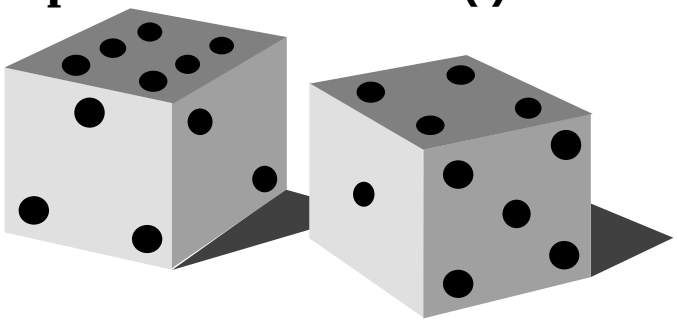
- `Date` need `#include "date.h"`
  - Objects: December 7, 1949, November 22, 1963
  - Methods: `MonthName()`, `DaysIn()`, `operator -`

Anatomy of the Dice class

- The class Dice, need `#include "dice.h"`
  - Objects: six-sided dice, 32-sided dice, one-sided dice
  - Methods: `Roll()`, `NumSides()`, `NumRolls()`

- A Dice object has state and behavior
  - Each object has its own state, just like each int has its own value
    - Number of times rolled, number of sides
  - All objects in a class share method implementations, but access their own state
    - How to respond to `NumRolls()`? Return my own # rolls





The header file dice.h

```
class Dice
{
  public:
    Dice(int sides);          // constructor
    int Roll();               // return the random roll
    int NumSides() const;     // how many sides
    int NumRolls() const;     // # times this die rolled

  private:
    int myRollCount;          // # times die rolled
    int mySides;              // # sides on die
};
```

- The compiler reads this header file to know what's in a `Dice` object
- Each `Dice` object has its own `mySides` and `myRollCount`

The header file is a class *declaration*

- What the class looks like, but now how it does anything
- Private data are called *instance variables*
  - Sometimes called data members, each object has its own
- Public functions are called *methods*, *member functions*, these are called by *client* programs

- The header file is an interface, not an implementation
  - Description of behavior, analogy to stereo system
  - Square root button, how does it calculate? Do you care?

- Information hiding and encapsulation, two key ideas in designing programs, object-oriented or otherwise

From interface to use, the class Dice

```
#include “dice.h”
int main()
{
    Dice cube(6);
    Dice dodeca(12);
    int x = cube.Roll();
    int k;
    for(k=0; k < 6; k++)
    {   x = dodeca.Roll();
    }
    return 0;
}
```

Objects constructed

```
cube.myRollCount == 0
cube.mySides == 6
dodeca.myRollCount == 0
dodeca.mySides == 12
```

Method invoked

```
cube.myRollCount == 1
cube.mySides == 6
```

After for loop

```
dodeca.myRollCount == 6
dodeca.mySides == 12
```

Header file as Interface

- **Provides information to compiler and to programmers**
  - **Compiler determines how big an object (e.g., Dice cube(6)) is in memory**
  - **Compiler determines what methods/member functions can be called for a class/object**
  - **Programmer reads header file (in theory) to determine what methods are available, how to use them, other information about the class**
- **What about CD, DVD, stereo components?**
  - **You can use these without knowing how they really work**
  - **Well-designed and standard interface makes it possible to connect different components**
  - **OO software strives to emulate this concept**

William H. (Bill) Gates, (*b. 1955*)

- **CEO of Microsoft, richest person in the world (1999)**
  - **First developed a BASIC compiler while at Harvard**
  - **Dropped out (asked to leave?) went on to develop Microsoft**

"You've got to be willing to read other people's code, then write your own, then have other people review your code"

- **Generous to Computer Science and philanthropic in general**
- **Visionary, perhaps cutthroat**

From Interface to Implementation

- **The header file provides compiler and programmer with how to use a class, but no information about how the class is implemented**
  - **Important separation of concerns, use without complete understanding of implementation**
  - **Implementation can change and client programs won't (hopefully) need to be rewritten**
    - **If private section changes, client programs will need to recompile**
    - **If private section doesn't change, but implementation does, then client programs relinked, but not recompiled**

- **The implementation of foo.h is typically in foo.cpp, this is a convention, not a rule, but it's well established (foo.cc used too)**

Implementation, the .cpp file

- **In the implementation file we see all member functions written, same idea as functions we've seen so far**
  - **Each function has name, parameter list, and return type**
  - **A member function's name includes its class**
  - **A *constructor* is a special member function for initializing an object, constructors have no return type**

```
Dice::Dice(int sides)
// postcondition: all private fields initialized
{
    myRollCount = 0;
    mySides = sides;
}

int Dice::NumSides() const
// postcondition: return # of sides of die
{
    return mySides;
}
```

More on method implementation

- Each method can access private data members of an object, so same method implementation shared by different objects
  - `cube.NumSides()` compared to `dodeca.NumSides()`

```
int Dice::NumSides() const
// postcondition: return # of sides of die
{
    return mySides;
}

int Dice::Roll()
// postcondition: number of rolls updated
//                random 'die' roll returned
{
    RandGen gen;    // random number generator ("randgen.h")

    myRollCount= myRollCount + 1;    // update # of rolls
    return gen.RandInt(1,mySides);   // in range [1..mySides]
}
```

Understanding Class Implementations

- **You do NOT need to understand implementations to write programs that use classes**
  - **You need to understand interfaces, not implementations**
  - **However, at some point you'll write your own classes**

- **Data members are global or accessible to each class method**
- **Constructors should assign values to each instance variable**

- **Methods can be broadly categorized as *accessors* or *mutators***
  - **Accessor methods return information about an object**
    - **`Dice::NumRolls()` and `Dice::NumSides()`**
  - **Mutator methods change the state of an object**
    - **`Dice::Roll()`, since it changes an object's `myNumRolls`**

Class Implementation Heuristics

- **All data should be private**
  - **Provide accessor functions as needed, although classes should have more behavior than simple `GetXXX` methods**
- **Make accessor functions `const`**
  - **Easy to use `const` functions (we'll see more on `const` later), although difficult at times to implement properly**
  - **A `const` function doesn't modify the state of an object**

```
int Dice::NumSides() const
// postcondition: return # of sides of die
{
    return mySides;
}
```

Building Programs and Classes

- **To develop a program, written with classes or not, start small**
  - **Get a core working, and add to the core**
  - **Keep the program working, easier to find errors when you've only a small amount of new functionality**
  - **Grow a program incrementally rather than building a program all at once**

- **Start with a prototype**
  - **Incomplete, but reasonable facsimile to the final project**
  - **Help debug design, ideas, code, …**
  - **Get feedback to stay on track in developing program**
    - **From users, from compiler, from friends, from yourself**

Design Heuristics

- **Make each function or class you write as single-purpose as possible**
  - **Avoid functions that do more than one thing, such as reading numbers and calculating an average, standard deviation, maximal number, etc.,**
    - **If source of numbers changes how do we do statistics?**
    - **If we want only the average, what do we do?**
  - **Classes should embody one concept, not several. The behavior/methods should be closely related**

- **This heuristic is called *Cohesion*, we want functions and classes to be cohesive, doing one thing rather than several**
  - **Easier to re-use in multiple contexts**

Design Heuristics continued

- **Functions and classes must interact to be useful**
  - **One function calls another**
  - **One class uses another, e.g., as the `Dice::Roll()` function uses the class `RandGen`**

- **Keep interactions minimal so that classes and functions don't rely too heavily on each other, we want to be able to change one class or function (to make it more efficient, for example) without changing all the code that uses it**

- **Some *coupling* is necessary for functions/classes to communicate, but keep coupling loose**
  - **Change class/function with minimal impact**

Reference parameters

- It's useful for a function to return more than one value
  - Find roots of a quadratic
  - Get first and last name of a user
- Functions are limited to one return value
  - Combine multiple return values in object (create a class)
  - Use reference parameters to send values back from function
    - Values not literally returned
    - Function call sends in an object that is changed
- Sometimes caller wants to supply the object that's changed

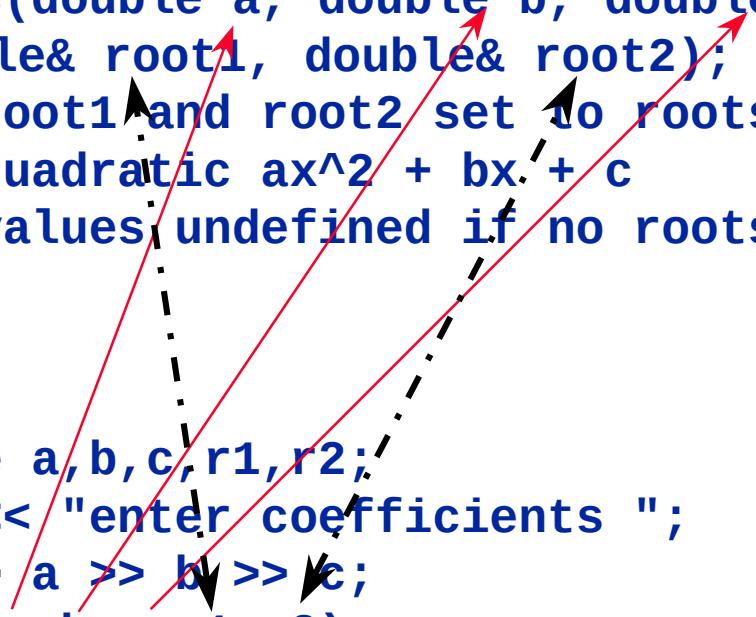
```
string s = ToLower("HEllO") // return type?
string s = "HeLLo";
ToLower(s);                 // return type?
```

Quadratic Equation Example

```
void Roots(double a, double b, double c,
      double& root1, double& root2);
// post: root1 and root2 set to roots of
//       quadratic ax^2 + bx + c
//       values undefined if no roots exist

int main()
{
    double a,b,c,r1,r2;
    cout << "enter coefficients ";
    cin >> a >> b >> c;
    Roots(a,b,c,r1,r2);

    cout << "roots are " << r1 << " " << r2 << endl;
    return 0;
}
```



Who supplies memory, where's copy?

```
void Roots(double a, double b, double c,
     double& root1, double& root2);
// post: root1 and root2 set to roots of
//       quadratic ax^2 + bx + c
//       values undefined if no roots exist
```

- For value parameter, the argument value is copied into memory that "belongs" to parameter
- For reference parameter, the argument is the memory, the parameter is an alias for argument memory

```
double x, y, w, z;
Roots(1.0, 5.0, 6.0, x, y);
Roots(1.0, w, z, 2.0, x);    // no good, why?
```

Examples

- **What's prototype for a function that rolls two N-sided dice Y times and returns the number of double 1's and double N's**

- **What's prototype for a function that returns the number of hours, minutes, and seconds in N seconds?**

- **What's prototype for a function that returns the number of Saturdays and the number of Sundays in a year?**

Parameter Passing: const-reference

- When parameters pass information into a function, but the object passed doesn't change, it's ok to pass a copy
  - Pass by value means pass a copy
  - Memory belongs to parameter, argument is copied
- When parameter is altered, information goes out from the fucntion via a parameter, a reference parameter is used
  - No copy is made when passing by reference
  - Memory belongs to argument, parameter is alias

- Sometimes we want to avoid the overhead of making the copy, but we don't want to allow the argument to be changed (by a malicious function, for example)
  - *const-reference* parameters avoid copies, but cannot be changed in the function

Count # occurrences of “e”

- Look at every character in the string, avoid copying the string

```
int LetterCount(const string& s, const string& letter)
// post: return number of occurrences of letter in s
{
    int k, count = 0, len = s.length();
    for(k=0; k < len; k++)
    {    if (s.substr(k,1) == letter)
         {    count++;
         }
    }
    return count;
}
```

- Calls below are legal

```
int ec = LetterCount("elephant", "e");
string s = "hello"; cout << LetterCount(s, "a");
```

General rules for Parameters

- Don't worry too much about efficiency at this stage of learning to program
  - You don't really know where efficiency bottlenecks are
  - You have time to develop expertise
- However, start good habits early in C++ programming
  - Built-in types: int, double, bool, char, pass by value unless returning/changing in a function
  - All other types, pass by const-reference unless returning/changing in a function
  - When returning/changing, use reference parameters

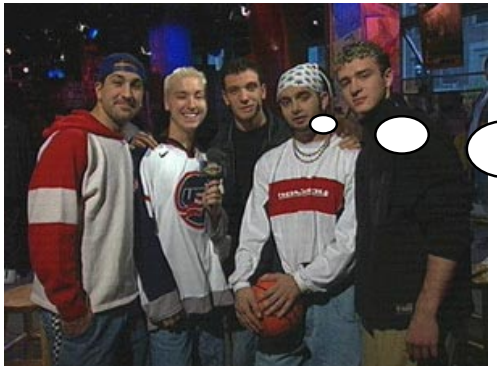
- Const-reference parameters allow constants to be passed, "hello" cannot be passed with reference, but ok const-reference

Rock Stars for Computer Science

I was going to call it “Songs in the Key of C++”

I love to program biorhythms

I think about C++ while I’m driving



We’d tell you about programming, but our brains are empty

Streams for reading files

- We've seen the standard input stream `cin`, and the standard output streams `cout` (and `cerr`)
  - Accessible from <iostream>, used for reading from the keyboard, writing to the screen

- Other streams let us read from files and write to files
  - Syntax of reading is the same: a stream is a stream
  - Syntax for writing is the same: a stream is a stream

- To use a file stream the stream must be opened
  - Opening binds stream to a file, then I/O to/from is ok
  - Should close file streams, but happens automatically

Input file stream: Note similarity to cin

```
string word;
int numWords = 0;        // # words read so far
while (cin >> word)      // read succeeded
{   numWords++;
}
cout << "number of words read = " << numWords << endl;


ifstream input;          // need <fstream> for this
string filename = PromptString("enter name of file: ");
input.open(filename.c_str());

while (input >> word)    // read succeeded
{   numWords++;
}
cout << "number of words read = " << numWords << endl;
```

Find longest word in file (most letters)

- **Idea for algorithm/program**
  - **Read every word, remember the longest word read so far**
  - **Each time a word is read, compare to longest-so-far, if longer then there's a new longest-so-far**

- **What should longest-so-far be initialized to?**
  - **Short word? Long word? First word?**

- **In general, when solving *extreme-value* problems like this use the first value as the initial value**
  - **Always works, but leads to some duplicate code**
  - **For some values a default initialization is possible**

Maximal and Minimal Values

- Largest int and double values are found in `<climits>` and `<cfloat>`, respective (`<limits.h>` and `<float.h>`)
  - `INT_MAX` and `INT_MIN` are max and min ints
  - `DBL_MAX` and `DBL_MIN` are max and min doubles

- What about maximal string value alphabetically? Minimal?
  -

- What about longest string in length? Shortest?
  -

Using classes to solve problems

- **Find the word in a file that occurs most frequently**
  - **What word occurs most often in *Romeo and Juliet***
  - **How do we solve this problem?**

- **Suppose a function exists with the header below:**

```
int CountOccurrences(const string& filename,
                     const string& s)
// post: return # occurrences of s in file w/filename
```

- **How can this function be called to find the maximally occurring word in *Romeo and Juliet*?**
  - **Read words, update counter?**
  - **Potential problems?**

Complete the code below

```
int CountOccurrences(const string& filename, const string& s)'
// post: return # occurrences of s in file w/filename

int main()
{
    string filename = PromptString("enter file: ");
    ifstream input(filename.c_str());
    string word;

    while (input >> word)
    {    int occs = CountOccurrences(filename, word);




    }
    cout << "maximally occurring word is " << maxWord << endl;
    cout << "which occurs " << max << " times" << endl;
}
```

Two problems

- **Words appear as The and the, how can we count these as the same? Other issues?**
  - **Useful utilities in "strutils.h"**
    - **ToLower** **return a lowercase version of a string**
    - **ToUpper** **return an uppercase version of a string**
    - **StripPunc** **remove leading/trailing punctuation**
    - **tostring(int)** **return "123" for 123, int-to-string conversion**
    - **atoi(string)** **return 123 for "123", string-to-int conversion**

- **We count occurrences of "the" as many times as it occurs**
  - **Lots of effort duplicated, avoid using the class `StringSet`**
  - **A set doesn't store duplicates, read file, store in set, then loop over the set counting occurrences**

StringSet and WordIterator

- **Both classes support access via *iteration***
  - **Iterating over a file using a `WordIterator` returns one word at-a-time from the file**
  - **Iterating over a set using a `StringSetIterator` returns one word at-a-time from the set**

- **Iteration is a common pattern: A pattern is a solution to a problem in a context**
  - **We'll study more patterns later**
  - **The pattern has a name, and it's an idea embodied in code rather than the code itself**

- **We can write code without knowing what we're iterating over if the supports generalization in some way**

See setdemo.cpp

```
#include <iostream>
using namespace std;
#include "stringset.h"

int main()
{
    StringSet sset;
    sset.insert("watermelon"); sset.insert("apple");
    sset.insert("banana");     sset.insert("orange");
    sset.insert("banana");     sset.insert("cherry");
    sset.insert("guava");      sset.insert("banana");
    sset.insert("cherry");

    cout << "set size = " << sset.size() << endl;

    StringSetIterator it(sset);
    for(it.Init(); it.HasMore(); it.Next())
    {   cout << it.Current() << endl;
    }
    return 0;
}
```