Announcements

- Recitations:
  - Next week: Google Test (unit testing harness)
  - Next week: Mandatory
- PA4 is due Wednesday
  \[ W[i,j] = D[i,j] + M\ln(W[i-1,k]), k \leq j \]
- Any questions?
- Expect a reading assignment for Wednesday

Welcome to CS253, Part 2

- Part 1 focused on memory
  - Stack vs heap
  - Scoping
  - Memory management
- Part 2 will focus on Object Oriented Programming in C++
  - Encapsulation
  - Abstraction via inheritance
    - Inheritance as union
    - How are objects implemented?
    - Multiple inheritance
  - Dispatch
  - Testing starts with recitation on GoogleTest
  - Overloading starts today

Unit Testing

- As programs get more complex...
  - End-to-end testing comes last
    - For most of you, this is the only testing you’ve done
  - Designing test cases comes first
    - Before you write any code
    - Include typical cases
    - Include boundary cases
  - Unit testing happens during development
    - Write a test for every public method you define
    - Test it independent of the larger system
    - Test for its boundary cases
      - Even if they don’t come up in the larger system (you think)
      - You might want to re-use your code someday

Overloading

- C++ allows you to define functions (or methods) for operators on non-primitive data types.
- Example:
  - I create a Complex number class
  - I want complex numbers to behave
    - Like other numbers
    - Like they do in math textbooks
  - So I overload +, *, /, etc.
  - Now Complex numbers act like primitive numbers
- Java doesn’t do allow operator overloading
  - You cannot define a class that works like a primitive
  - Of course, you can’t pass an object like a primitive, either.

Syntax of Overloading

- Like any other method or function, except
  - Name is operator op
  - Where ‘operator’ is a keyword
  - Where op is the operator symbol being overloaded
Example of Overloading

```cpp
class Complex {
public:
    Complex(double real_part = 0.0, double imaginary_part = 0.0)
        : real(real_part), imaginary(imaginary_part) {}      
inline double Real() const {return real;}               
inline double& Real() {return real;}                    
inline double Imaginary() const {return imaginary;}   
inline double& Imaginary() {return imaginary;}         
    void Add(const Complex& add_in);
    void Subtract(const Complex& sub_in);
    void Multiply(const Complex& mult_in);
    bool Divide(const Complex& div_in);

    Complex operator + (const Complex& in) const {
        Complex out(Real() + in.Real(), Imaginary() + in.Imaginary());
        return out;
    }

    Complex operator - (const Complex& in) const {
    }
    Complex operator * (const Complex& in) const {
    }
    Complex operator / (const Complex& in) const {

};
```

What can be overloaded?

- Math:
  - + - * / % == != -= /= += %=
- I/O:
  - << >>
- Logic:
  - & l ! ~ = <<= >>= ^ ^= & &=
- Indexing:
  - () [] ++ -- . -> * new delete new[] delete[
- You cannot overload
  - .* ? sizeof

Limitations on overloading

- You cannot change the arity of an operator
  - += takes two arguments.
  - + can take one arguments (as in a = +4)
  - + can take two arguments (as in a = 4+4)
  - You cannot define a 3 argument +
  - You cannot define a 1 argument +=
- You cannot change operator precedence
  - In a+bc, * happens before +
- General rule: you cannot change how the language is parsed

Functions vs Methods

- You can define an operator to be a method
  - As in the complex example
  - Or a function
  - Define it outside of any class
- If you define it as a method
  - Remember the hidden argument
  - += as a method takes one explicit argument (and the hidden argument)
  - Methods have access to protected data
- I use functions when
  - The operator is symmetric (e.g. add)
  - The data is publicly accessible
  - The arguments are of different types
  - There are no side-effects
- Otherwise, I use methods

What about Java?

- Java doesn’t support operator overloading
- Why not?
  - Java is afraid you will misuse/overuse it
  - Programmers have very strong intuitions about operators
    - We all know what + means
    - We all know what << means
  - Overloading operators with functions that don’t match these intuitions creates confusion
    - Better not to use it than to use it in the wrong case
    - Also, Java doesn’t have templates
**When should I overload an operator?**

- When a method exactly matches the expectations we have about an operator
- Most common: I/O
  - If you write an class that is written or read
    - To files
    - To the screen
  - Then you should overload `<<` and/or `>>`
  - Makes sense for data bases, too

**Example Functor**

```cpp
class Add3 {
public:
  int operator()(int value) const
  { return value+3; }
};

int main(int argc, char* argv[])
{
  Add3 add3;
  // add3 is now an object I can pass...
  return add3(-2);
}
```

**When (continued)**

- 2nd most common case: Functors
  - C programs sometimes pass function pointers
  - In C++ it is better to
    - Define a class with a method that performs the function
    - Overload operator () for the class
    - Pass the object

**Note that functors have local memory...**

```cpp
class AddN {
public:
  AddN(int n) : incr(n) {}
  int operator()(int value) const
  { return value+incr; }
  protected:
    int incr;
};

int main(int argc, char* argv[])
{
  AddN add(5);
  return add(-4);
}
```

**When (continued)**

- Less common case: Numerics
  - Mathematical objects support math operators
  - Examples:
    - Matrix
    - Tensor
    - Rational
    - Complex
    - Angle
  - Sometimes logic operators, but iffier
    - Streams overload `!`, for example