Intermediate C++

getline()

- Need to read an entire line into a string? Use getline():

```
istream& getline(istream  &in, string &target)
```

- reads from current location to end of line from input stream in
- places result into string target
- WARNING: includes whitespace.

getline() example

- Suppose you have an input file like:

```
1234
hello world, how are you?
```

- The following code:

```
getline(inputStream, str1);
getline(inputStream, str2);
```

  - reads “1234” into str1
  - “hello world, how are you?” into str2

getline() example

- BE CAREFUL!
- Suppose you have an input file like:

```
1234
hello world, how are you?
```

- The following code:

```
int val;
inputStream >> val;
getline(inputStream, str2);
```

  - reads 1234 into val
  - “” into str2 — yes, really!

Operator Overloading

- C++ allows programmers to modify (or add) operator behavior.
- Consider a Complex Number Class:

```
class Complex
{
private:
  double real;
  double imaginary;
public:
  // many methods skipped …
  Complex add(Complex otherNumber);
}
```

- using the add method with Complex variables a,b,c to do c=a+b:

```
  c = a.add(b);
```

Operator Overloading…

- Rewrite the previous class by renaming the add method:

```
class Complex
{
private:
  double real;
  double imaginary;
public:
  // many methods skipped …
  Complex operator+(Complex otherNumber);
}
```

- using the operator+ method with Complex variables a,b,c to do c=a+b:

```
c = a.operator+(b);
```

- or better yet …

```
c=a+b;
```

Operator Overloading…

- Now consider the ability to also add a floating point value:

```
class Complex
{
private:
  double real;
  double imaginary;
public:
  // many methods skipped …
  Complex operator+(Complex otherNumber);
  Complex operator+(double otherNumber);
}
```

- using Complex variables a,b and double d to do b=a+d

```
a= b.operator+(d);
```

- or better yet …

```
a= b+d;
```

Operator Overloading…

- But, what if we wanted to do (using Complex variables a,b and double d)

```
a= d+b;
```

- Unfortunately, this gives a compiler error, actually trying:

```
a = d.operator+(b);
```

- d is of type double, which is not an object
  - cannot do object-dot-method if left side of "." is not an object!!
  - also, d is not of type Complex, so our method does not apply!

Operator Overloading via friends…

- Instead, write a non-member function:

  Complex operator+(double rval, Complex cval);

  - note that this function takes two arguments (LHS & RHS)!
- Unfortunately, you likely need access to private instance variables (i.e. cval's instance variables)
- C++ allows a class to declare non-member function to be a "friend":
  - inside the class, give a prototype for the function
  - **and** prefix it with the keyword friend:

```
class Complex
{
   // … lots skipped here …
   friend Complex operator+(double rval, Complex cval);
}
```

The **const** keyword

- the **const** keyword in C++ indicates that either
  - the associated item cannot be modified or
  - the associated item may not make modifications

- const can show up in many places in C++

- ex. global (not instance) “variables”:

```
const double PI=3.14159;
const double E=2.7183;
```

The **const** keyword…

- ex. at the end of a method header:
  - the method cannot modify the associated object
  - used for accessor methods
  - ex. (inside the Complex class)

```
Complex operator+(Complex other) const;
```

  - means that operator+ will not modify “this” Complex number.

The **const** keyword…

- ex. in front of a parameter:
  - the method cannot modify the associated parameter

  - ex. (inside the Complex class)

```
Complex operator+(const Complex other) const;
```

  - means that operator+ will not modify the “other” Complex number.
  - (the const at the end still means that operator+ will not modify “this” Complex number).

The **const** keyword…

- ex. in front of a parameter (continued discussion):
  - the method cannot modify the associated parameter
    - even if the parameter is passed by reference!
  - ex. (inside the Complex class)

```
Complex operator+(const Complex &other) const;
```

  - means that operator+ will not modify the "other" Complex number.
  - The above is a common C++ technique.
    - saves execution time in parameter passing
    - still ensures that the parameter value does not change!

Return By Reference

- An & can be placed with a return value, ex.

```
double& imaginary(); // returns imaginary part
```

- Returns a **reference to** the return value
- Allows method call on LHS of assignment, ex.

```
Complex num(2,3); // num is 2+3i
num.imaginary() = 8; // num is now 2+8i
```

- No changes needed to code such a method, ex:

```
double&
Complex::imaginary() { return _imaginary;}
```

Return By Reference ...

- Primary use is to allow LHS usage in assignment
- Can also be used with const keyword:

```
const Complex& doSomething();
```

- Saves execution time when returning large object

Return By Reference Issue

- return by reference **refers** to the variable returned.

```
const int& simpleExample(int x, int y)
{
    int sum = x+y;
    return sum;  // returns reference to variable sum
}
```

- The above code contains subtle (run time) error
  - a **reference** to sum (a local) was returned …
  - … but sum has gone out of scope!!!
  - called "returning a reference to a local"
    - **very** common C++ coder error. Avoid it at all costs!