Operator Overloading

- The same operator represents different functions depending on the data type of its first operand.

  E.g.   a + b

  o   means integer addition when variable a is an int

  o   means floating point addition when variable a is a float

  E.g.   a >> b

  o   means stream extraction when variable a is a file stream

  o   means right shift when variable a is an unsigned int

- C++ allows the conventional operators (e.g. +, -, +=, >>, etc.) to be overloaded.

- To overload an operator function in class:

  operator   op(single-argument)

  The expression

  a op b   means   a.op(b)

A Complex Number Class with Operators

```
// FILE: complex.h
//   Defines complex number class

class complex
{
  public:
    // constructors
    complex();
    complex(float, float);

    // get real and imaginary part
    float get_real();
    float get_im();

    // operations
    complex operator +(const complex&);
    complex operator *(const complex&);
    ...

  private:
    float real;
    float im;
};
```

```
// FILE: complex.cpp
//   Complex number class implementation
#include "complex.h"
// Constructors
complex::complex() {}
complex::complex(float x, float y)
{
  real = x;
  im = y;
}
// get real and imaginary part
float complex::get_real()
{
  return real;
}
float complex::get_im()
{
  return im;
}
// Operations
complex complex::operator +(const complex& x)
{
  complex result;
  result.real = real + x.real;
  result.im = im + x.im;
  return result;
}
complex complex::operator *(const complex& x)
{
  complex result;
  result.real = real*x.real - im*x.im;
  result.im = real*x.im + im*x.real;
  return result;
}
```

Example:

```
#include "complex.h"
int main()
{
   complex x, y, z1, z2;

   x = complex(2.0, 1.0);
   y = complex(1.0, -3.0);

   z1 = x + y;
   z2 = x * y;

   return 0;
}
```

Software Life Cycle

- Typical development steps of large and complex software:

1. Requirement specifications

o Prepare a complete and unambiguous problem statement or software specification

2. Problem analysis

o Understand the problem: input/output requirements, performance requirements and constrants.

o Evaluate alternative solutions

o Choose a preferred solution

3. Program design

o High level design

- Perform a top-down design of the system
- Specify the overall software structure
- For each module, specify its interface with other modules

o Low level design

- Algorithm design for each low-level module, using pseudo-code

4. Program implementation

o Code the solution according to the design.

5. Testing and validation

o Test individual module

o Integration testing, validate correstness according to specification

6. **Operation**
   - Run the completed system
   - Evaluate performance

7. **Possible enhancements and maintenance**
   - Remove the reported bugs
   - Update software specification
   - Make required changes to keep the system up-to-date
   - Fully test and validate the system after any new changes

This is roughly the flow of development. In practice, there are usually many iterations and revisions.