Unit 0.2 C++ Review

Numerical Analysis

EE/NTHU

Mar. 5, 2020

Mathematical Operations in C and C++

- In C or C++ we have the following basic numerical types defined already.
  - Integer types: `int`, `short`, `long`, `unsigned`, `singed`, etc.
  - Floating-point types: `float`, `double`, `long double`.
- Using these basic types, mathematical expressions can be coded using simple operators.
  - For example, to evaluate

$$y = \frac{x^2+1}{x^3+2x^2+3x+4}.$$

- One simply writes

```
y = (x*x + 1) / (x*x*x + 2*x*x + 3*x + 4);
```

- Thus, simple mathematical expressions can be coded with ease.
- This is because the operators associated with the basic types have been defined.

Needs of Vector and Matrix Operations

- In this course, we will need to solve linear and nonlinear systems.
- Often these systems are expressed in matrices and vectors.
- Solution methods are often expressed using matrices and vectors as well.
- The vectors and matrices are not basic types in C or C++.
- And no mathematical operations defined.
- For example, given an $n \times n$ matrix $\mathbf{A}$, an n-vector $\mathbf{x}$ and a real number λ, if λ is an eigenvalue of $\mathbf{A}$ and $\mathbf{x}$ is the associated eigenvector, then

$$\mathbf{Ax} - \lambda\mathbf{x} = \mathbf{0}. \tag{0.2.1}$$

- Vectors are usually defined as 1-dimensional arrays in C or C++; while matrices defined as 2-dimensional arrays. To check if Eq. (0.2.1) is satisfied, multiple loops are needed.
- Fortunately, using class in C++ one can defined mathematical operators to match mathematical equations.
- Translating numerical theories to programs becomes an easy job.

Classes in C++

- Class in C++ is designed to provide programmers with a tool for creating new types that can be used as conveniently as the basic types.
- It needs to define two aspects: data stored and the operations associated with.
  - Data members to store necessary data for the new type.
  - Function members to define the operations.
- Data members are similar to struct in C.
  - Storage for any predefined types can be declared.
- The members of a class are further classified into two categories:
  - public: members can be accessed by any functions,
  - private: members can only be accessed by member functions.
- Using private data, data hiding is made possible.
  - Clear responsibility,
  - Easier debugging,
  - Enable better teamwork, etc.

Class Example

```
class Complex {
  public:
    Complex(double r=0, double i=0);    // constructor
    Complex(const Complex &C);          // copy constructor
    double r() const;                   // get real part
    double i() const;                   // get imaginary part
    Complex& operator+=(Complex &C2);   // C1 += C2;
    Complex& operator+=(double);        // C1 += dbl;
    Complex& operator-=(Complex &C2);   // C1 -= C2;
    Complex& operator-=(double);        // C1 -= dbl;
    Complex& operator*=(Complex &C2);   // C1 *= C2;
    Complex& operator*=(double);        // C1 *= dbl;
    Complex& operator/=(Complex &C2);   // C1 /= C2;
    Complex& operator/=(double);        // C1 /= dbl;
  private:
    double x,y;
};
```


Classes in C++, II

- Data members can be divided into public and private.
  - Data accessible by all functions should be declared public,
  - Other data declared private.
- Member functions can be either public or private as well.
  - Most member functions are public to define operations on the class.
  - Private functions can be used by member functions only.
    - Utility functions.
- The struct in C is a special case of class in C++.
  - It has only public members.
  - And no member functions.
- In declaring a class, the member function should be declared.
- All members (data and function) are accessed through the member operator `.` or `->`.
  - Accessing data: `z1.x`, `z1.y`,
  - Accessing function: `z1.r()`, `z1.i()`,
  - If zp is a pointer to z, then
    - Accessing data: `zp->x`, `zp->y`,
    - Accessing function: `zp->r()`, `zp->i()`.

Member Function Accessing

- In C or C++, a function, `f`, is invoked by
  - `f(x)`, calling `f` with one argument.
  - `f(x1, x2)`, calling `f` with two arguments.
  - `f()`, calling `f` with no argument.
  - The parentheses `()` are needed to signify a function call.
- When calling a member function, (assuming `z` is an Complex object)
  - `z.r()` or `z.i()`.
  - Parentheses are still needed to signify a function call.
  - There is an implicit argument supplied, `this`.
  - `this` is a pointer pointing to the object.
  - In `z.r()`, `this = &z`.
  - In the function definition of `r()`, data member can be accessed by the name of the data or with `this->` preceding the name of the data.
    - For example, the member function `r()` is defined as

```
double Complex::r()
{
    return x;
}
```

```
double Complex::r()
{
    return this->x;
}
```


Member Function Definition

- To define a member function, the name should be preceded with `class name::`, as the examples shown above.
  - The `return type` needs to precede the `class name`.
  - Note that in C++, the return type should always be defined.
- Two special functions need no return type.
  - constructor and destructor functions.
- In C or C++, a variable is created by a declaration statement.
  - For example, two variables are declared

```
double x1, x2;
```

- In C++, an object is instantiated by declaration as

```
Complex z1;
```

When this declaration statement is executed, the constructor function is actually called such that the data member can be initialized (or not) properly.

Constructors

- Three types of constructors are possible (using Complex class as an example).
  - Initialization constructor:
    `Complex z1(1.0, 1.0);    // initializing z1 to (1.0, 1.0)`
  - Uninitialization constructor, two usages:
    `Complex z2;              // uninit constructor`
    `Complex z2();            // uninit constructor`
  - Copy constructor, two usages:
    `Complex z3 = z1;         // copy constructor`
    `Complex z3(z1);          // copy constructor`
- Each constructors need to declared and defined separately.

```
Complex::Complex(double r,double i)  // init constructor
{
    x = r; y = i;
}
Complex::Complex()               // uninit constructor, no actions required
{ }
Complex::Complex(const Complex &z)   // copy constructor
{
    x = z.x; y = z.y;
}
```


Constructors, II

- Note that the name of the constructor functions are simply the name of the class.
  - In defining the member function, class name and scope operator `::` need to precede function name.
- Constructor functions need no return type.
- Three functions defined with the same name – function overload.
  - The number of arguments or the argument types need to be different.
- In C++, function arguments can have default values.

```
Complex::Complex(double r=0, double i=0)   // init constructor
{
    x = r; y = i;
}
```

- In this case, when parameters are not supplied in function calls, the default values are taken.

```
    Complex z1(1.0, 1.0);      // initialize both real and imaginary
    Complex z2(1.0);           // z2 = (1.0, 0)
    Complex z3;                // z3 = (0, 0)
```

- In this case, the uninit constructor should not be declared or defined.

Destructor

- Destructor function is called when a variable is no longer needed.
  - Local variables (automatic storage duration) going out of scope.
  - Temporary variables no longer needed.

```
z4 = z1 + z2 * z3;
z5 = sqrt(z6);
```

  A temporary variable may be created in forming the product `z2*z3`, and returned by a function before assignment.

- Compiler generates calls to destructor functions.
- In most cases, destructor needs no actions.
  - But, if dynamic memory is allocated by constructor then destructor should release the memory space acquired.
- The name of destructor function is $\boxed{\sim}$ preceding the class name.
- Destructor has no return type.

```
Complex::~Complex()
{ }                                       // no action needed
```


References

- In C, function parameters are passed by value.
  - A copy of the variable is created and passed to the function.
  - The return value of a function is also a copy that is destroyed after its use.
  - Thus, the value of the parameter cannot be modified by a function.
  - Pointer should be passed, if its value is to be modified.
- In C++, the variable itself can be passed and returned by a function – pass-by-reference.
- To do so, the parameter needs to be declared with a reference symbol $\boxed{\&}$ after the type name.
- Example member function declaration:

```
Complex& operator+= (Complex &C2);  // C1 += C2;
```

- Example member function definition:

```
Complex & Complex::operator+=(Complex &z)
{
    x += z.x;
    y += z.y;
    return *this;
}
```

References, II

- Using a referenced parameter or return, the variable itself is passed between functions.
  - No copies are created – improved efficiency.
  - Variable can be modified.
- Reference can be used to create an alias for a variable.

```
Complex z1;
Complex &za = z1;
```

The variable za is an alias of z1. Modifying za changes the value of z1, vice versa.

- Note that reference can only be created for variables that are existing physically. Temporary variables created during expression evaluation may not have references.

```
z4 = z1 + z2 * z3;
```

  - The temporary variable created for z2*z3 may not be physically present and, thus, no reference possible.

const

- Using reference can increase program efficiency by eliminating the need of copying function arguments repeatedly.
- The function is also able to access the variables with the capability of modifying their values.
- To clearly state that a function argument will not be modified, the `const` key word should be used.

```
Complex& operator+=(const Complex &C2);  // C1 += C2;
```

- With `const` inserted, the variable `C2` cannot be modified.
- Any attempts of modifying `C2` will result in a compilation error.
- Note that the copy constructor should have the argument declared as a `const reference` of the class type.

```
class Complex {
    // ...
    Complex(const Complex &C);     // copy constructor
    // ...
}
```

const, II

- Note that const can be used for any argument, not necessary of reference type, with the same effect.
- A member function of a class that will not modify any of its data members can be declared to be a const member function.

```
class Complex {
    // ...
    double r() const;          // return real part
    double i() const;          // return imaginary part
    // ...
}
double Complex:r() const
{
    return x;
}
```

- Note that in function definition, the const key work needs to appear after the parentheses as well.

Defining Operators for Classes

- In C++, most operators can be defined for classes
  `+` `-` `*` `/` `%` `^` `&` `|` `~` `!` `=` `<` `>` `+=` `-=` `*=` `/=` `%=`
  `^=` `&=` `|=` `<<` `>>` `>>=` `<<=` `==` `!=` `<=` `>=` `&&` `||` `++` `--`
  `->*` `,` `->` `[]` `()` `new` `new[]` `delete` `delete[]` .
- The following operators cannot be redefined.
  - `::` scope resolution,
  - `.` member selection,
  - `.*` member selection through pointer to member,
  - `? :` ternary condition expression,
  - `sizeof`,
  - `typeid`.
- The first 3 operators take a name, rather than value, as the second operand.

Defining Operators for Classes, II

- In C++, the operators are functions.
  - They have the name `operator` preceding the `operator symbol`.
  - For example, `operator+`.
- If z is an object of class Complex and `operator+` function is defined as a member function of class Complex, then it is possible to use both shorthand or explicit operator function call.
- If `operator+` is defined as a member function of z1.

```
z3 = z1 + z2;
z3 = z1.operator+(z2);
```

- If `operator-` is defined as a nonmember function, then the following are equivalent.

```
z3 = z1 - z2;
z3 = operator-(z1, z2);
```

- Shorthand operator symbols are usually preferred.

Defining Operators for Classes, III

- In defining an operator function as a member function, it should be declared in the class declaration as well, and then defined with the class name preceding the function name.

```
class Complex {                        // class declaration
  public:
   // ...
   Complex operator+(Complex C);       // + function declaration
   // ...
};
Complex Complex::operator+(Complex C)  // + function definition
{ ... }
```

- In defining operator function as nonmember function, it follows regular function declaration and definition.

```
Complex operator-(Complex C1, Complex C2);  // - function declaration

Complex operator-(Complex C1, Complex C2)   // - function definition
{ ... }
```

Defining Operators for Classes, IV

- Note in the preceding function declaration and definition, the addition of `z1+z1` must have both operands of type Complex.
- If `z1` or `z2` is not Complex, then the addition is not defined.
- To handle the case that `z2` is a double, the function needs to be overloaded by declare and define the following:

```
class Complex {                        // class declaration
  public:
   // ...
   Complex operator+(double dbl);      // + can handle Complex + dbl
   // ...
};
Complex Complex::operator+(double dbl)  // + function definition
{ ... }
```

- But the case of *double* `z1` can only be handled by declaring a nonmember function.

```
    Complex operator+(double dbl, Complex C);
```

- This is due to that the first operand of the operator defined as member function must be an implicit argument, which has the same type as the class.

friend of a Class

- In `C++`, private members cannot be accessed by nonmember functions.
- Exceptions can be made by declaring a function as a friend.
  - `friend` functions can access private members (data and functions).
  - `friend` functions must be declared in class declaration.

```
class Complex {               // class declaration
  public:
   // ...
   friend double sqrt();      // function declaration
   // ...
};
```

- Note that `sqrt()` is a nonmember function but it can access class Complex's private data.
- Properties of `friendship`.
  - Friendship is granted not taken.
  - Friendship is not symmetric.
  - Friendship is not transitive.

Header Files and Source Files

- A well defined class can be reused for many applications.
- It is a good practice to put the declarations associated with a class into a header file that ends with `.h` file extension.
- The function definitions, member and nonmember, are put into a separated source file that ends with `.cpp` file extension.
- The source file can then be compiled by

```
$ g++ -c Complex.cpp
```

This produces a Complex.o object file that can be linked with application programs.

```
$ g++ prog.cpp Complex.o
```

- This practice saves compilation time and the header file serves as a well defined interface for the users.
- As long as the function declarations are not changed, modifying the the function definitions in Complex.cpp file does not affect the application source file.
  - But the program needs to be re-linked.

Complex.h (1/2)

```
// complex number class
#ifndef COMPLEX_H
#define COMPLEX_H
#include <stdio.h>
#include <stdlib.h>
class Complex {
  public:
    Complex(double r=0, double i=0);    // constructor
    Complex(const Complex &C);          // copy constructor
    double r() const;                   // get real part
    double i() const;                   // get imaginary part
    Complex& operator+=(Complex&);      // C1 += C2;
    Complex& operator+=(double);        // C1 += dbl;
    Complex& operator*=(Complex&);      // C1 *= C2;
    Complex& operator*=(double);        // C1 *= dbl;
    Complex& operator/=(Complex&);      // C1 *= C2;
    Complex& operator/=(double);        // C1 *= dbl;
  private:
    double x,y;
};
```

Complex.h (2/2)

```
Complex operator+(Complex);                 // unary plus
Complex operator+(Complex, Complex);        // C1 + C2
Complex operator+(Complex, double);         // C1 + dbl
Complex operator+(double, Complex);         // dbl + C1
Complex operator-(Complex);                 // unary minus
Complex operator-(Complex, Complex);        // C1 - C2
Complex operator-(Complex, double);         // C1 - dbl
Complex operator-(double, Complex);         // dbl - C1
Complex operator*(Complex, Complex);        // C1 * C2
Complex operator*(Complex, double);         // C1 * dbl
Complex operator*(double, Complex);         // dbl * C1
Complex operator/(Complex, Complex);        // C1 / C2
Complex operator/(Complex, double);         // C1 / dbl
Complex operator/(double, Complex);         // dbl / C1
double fabs(Complex z);                     // |C1|
Complex& operator++(Complex&);              // prefix increment
Complex operator++(Complex&, int);          // postfix increment
int operator==(Complex, Complex);           // testing for equality
int operator!=(Complex, Complex);
#endif
```


Complex.cpp (1/4)

```
// Function definitions
#include <math.h>
#include "Complex.h"
Complex::Complex(double r, double i)        // init constructor
{
    x = r; y = i;
}
Complex::Complex(const Complex &z)          // copy constructor
{
    x = z.x; y = z.y;
}
double Complex::r() const                   // get real part
{
    return x;
}
double Complex::i() const                   // get imaginary part
{
    return y;
}
```

Complex.cpp (2/4)

```
Complex & Complex::operator+=(Complex &z)  // C += Z;
{
    x += z.x;
    y += z.y;
    return *this;
}
Complex & Complex::operator+=(double d1)   // C += double;
{
    x += d1;
    return *this;
}
Complex operator+(Complex z)               // unary plus
{
    Complex z1(z);
    return z1;
}
Complex operator+(Complex z1, Complex z2)  // z1 + z2
{
    Complex z(z1);
    return z += z2;
}
```


Complex.cpp (3/4)

```
Complex operator+(Complex z1, double d)    // z1 + double
{
    Complex z(z1);
    return z += d;
}
Complex operator+(double d, Complex z1)    // double + z1
{
    Complex z(z1);
    return z += d;
}
double fabs(Complex z)                     // |z|
{
    return sqrt(z.r() * z.r() + z.i() * z.i());
}
Complex &operator++(Complex &z1)           // prefix increment
{
    return z1 += 1.0;
}
```

Complex.cpp (4/4)

```
Complex operator++(Complex &z1, int)        // postfix increment
{
    Complex z(z1);
    ++z1;
    return z;
}
int operator==(Complex z1, Complex z2)      // test for equality
{
    if (z1.r() == z2.r() && z1.i() == z2.i()) return 1;
    return 0;
}
```

- With these example functions, all functions defined in `Complex.h` can be easily implemented.
- And coding for complex number expressions are very straightforward.
- Try to code the following expressions.

$$f(z) = \frac{3z^3 + 2z^2 + z + 1}{z^4 - z^3 + z^2 - z + 1},$$
$$g(z) = (z+1)(z+2)(z+3).$$

Summary

- Needs of classes
- Class in C++
- Member functions
- Constructor
- Destructor
- References
- Const
- Defining operator functions
- Friends
- Header and source files
- Example: Complex class