ICOM 4035 – Data Structures

Dr. Manuel Rodríguez Martínez

Electrical and Computer Engineering Department

Lecture 4 – August 30, 2001

Readings

- Read chapter 2 of textbook
    - Objects and classes

Object-Oriented programming

- Objects
  - An instance of an entity of a give type
    - Ex. string name;
      - name represents an instance of an object of type string.
    - Its has:
      - Data members – variable that store some value/
      - Function members – functions (called methods) that implement the behavior of the object.
- Why do we need this?
  - Information hiding
    - User does not need to understand the internal details of the data structures and objects.
  - Encapsulation
    - Data data and operations that affect these data into an single entity.

O-O vs. procedural programming

- Object-Oriented Programming:
  1. Objects are treated as basic built-in types
  2. Information in an object can be hidden from programmer to prevent misuse.
  3. Methods associated with an object are members of the objects (part of its declaration and definition).

- Procedural programming:
  1. Structures are second class entities with restriction on their use.
  2. Information in a structure is available at any level to the programmer.
  3. Methods associated with an structure are independently declared and defined.

Classes in C++

- Class is the mechanism to create objects.
- They represent a user-defined data type
- Unlike structures, the members (data and methods) in a class are inaccessible to the general public
- Method Types:
  - Constructors – used to initialize the data members of an object.
  - Destructors – used to cleanup the data members in an object.
  - Accessors – used to read the values of the data members in an object.
  - Mutators – methods used to change the contents of data members (i.e. change the state of the object).

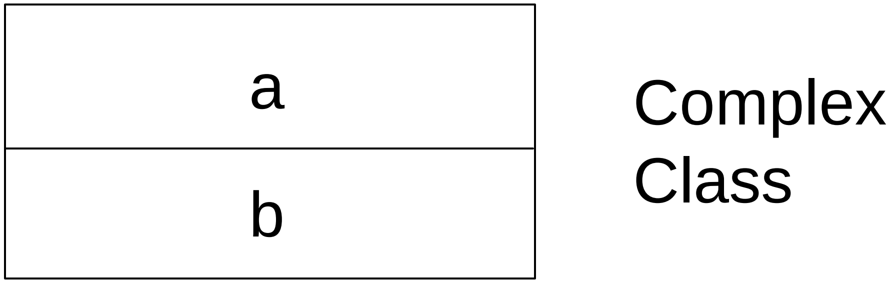
Example: Complex Numbers

- A complex number c has the form

  C = a + bi

  – Where, a is the real part of the complex number, and bi is the imaginary part. The coefficient b is a real number, and i is the imaginary number that represents the square root of –1.

- Initial design sketch:



a

b

Complex Class

  – No need to represent I!

Constructors

- Constructors – describes how an instance of the class is to created.
  - C++ provides default constructor that initializes members in a language dependent form.
- Called when a variable is initialized.
- Complex number constructors:
  - We need
    - one that initializes the number to 0.
    - one that initializes the number to a complex number based on a real part and a complex part.

Complex class and constructors

```
class complex {
    public:
        // empty arguments constructor
        complex(){
            real_part = 0;
            img_part = 0;
        }
        // constructor with arguments for complex number
        complex(double a, double b){
            real_part = a;
            img_part = b;
        }
    private:
        double   real_part; // the real part
        double   img_part; // the imaginary part
}
```

Public vs. private members

- Private members:
  - Only accessible by the routines that are member of the class.
  - Also accessible by the “friends” of the class.
  - In C++ everything is private by default.
- Public members:
  - Accessible by any routine in any class or anywhere in the program.
- Your APIs should be public, but your internal helper routines should be private.

Default parameters & initializer lists

- C++ provides mechanism to give default values to parameters of the constructors and functions
  - If the constructor is called with or more parameters not specified, then the defaults are used
  - Good way to consolidate multiple constructor declarations into just 1.
- Initializer lists are used to specify non-default initialization of data members.
  - Avoids creation of temporary objects to initialize complex objects.

Complex class: a refinement

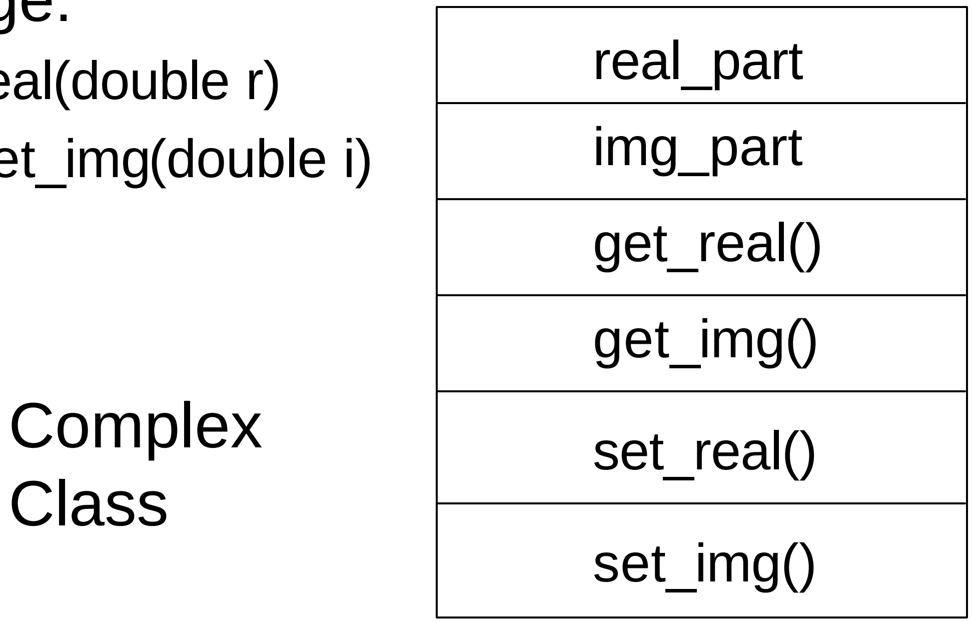
```
class complex {
    public:
        /*
         * new constructor has default parameters
         * plus initializer list.
         * We moved from 2 constructor into 1 that
         * covers all cases.
         * empty arguments constructor
         */
        complex(double a = 0, double b = 0)
            : real_part(a), img_part(b) {
            // empty body since all
            // initialization was done
        }
    private:
        double     real_part; // the real part
        double     img_part; // the imaginary part
}
```

Accessors and mutators

- Accessor is a method used to inspect the state of the object.
  - Access one or more data fields but don’t change them.
  - In C++ the keyword const is used at the end of method’s parameters list to indicate that it does not changes the state (I.e. it is an accessor)
- Mutator is a method used to change the state of the object.
  - Changes one or more data fields in the object.
- This is a good protocol to control the access to the data members of a class.

Complex class: refinement 2

- We need accessors to view:
  - Real part – get_real()
  - Imaginary part – get_img()
- Mutators to change:
  - Real part – set_real(double r)
  - Imaginary part- set_img(double i)



Complex
Class

real_part

img_part

get_real()

get_img()

set_real()

set_img()

Complex class: refined

```
class complex {
    public:
        // constructor with default values
        complex(double a = 0, double b = 0)
            : real_part(a), img_part(b) {
            // empty body since all
            // initialization was done
        }
        // accessors
        double  get_real(){return real_part;}
        double  get_img(){return img_part;}
        // mutators
        void set_real(double real){
            real_part = real;
        }
        void set_img(double img){
            img_part = img;
        }
    private:
        double       real_part; // the real part
        double       img_part; // the imaginary part
}
```

Class Interface and Implementation

- The class interface lists the class, its data members and its method members.
  - Tells us what can be done to a given object instance
- The class implementation represents the actual code that implements the behavior of the class as specified in the interface.
- You should separate them in different files!
  - Use .h file to put the interface
  - Use .cc, .ccp or .C file to put the implementation
    - Scope operator :: is used to indicate what class a method is associated with.
    - C++ convention:
      - class::method
      - class::datamember

Complex class: Interface

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H
class complex {
        public:
                // constructor with default values
                complex(double a = 0, double b = 0);
                // accessors
                double  get_real() const;
                double  get_img() const;

                // mutators
                void set_real(double real)
                void set_img(double img);
        private:
                double          real_part; // the real part
                double          img_part; // the imaginary part
}
#endif
```

complex.h file

Complex class: Implementation

```
#include “complex.h”

// constructor
complex::complex(double a, double b)
        : real_part(a), img_part(b){
        }
// accessors
double complex::get_real() const{
        return real_part;
}
double  complex::get_img() const{
        return img_part;
}

// mutators
void complex::set_real(double real){
        real_part = real;
}
void complex::set_img(double img){
        img_part = img;
}
```

complex.cc
file

Object declaration

- Objects are declared like primitive types
- Examples:
  - complex comp1; // initialized to complex number 0
  - complex comp2 (1); // initialized to complex number 1
  - complex comp3(0, 1); // initialized to complex number I
  - complex comp4(5, 9); // initialized to complex number 5 +9i

Operator overloading

- Extending the types to which the built-in operator can be applied in the program.
- Good way to make your user-defined type look like a built-in type
- Operator to overload:
  - +, -, *, /, %, +=, -=, *=, /=, %=, ++, - -, !=, ==, >, < , >=, <=, !, |, &, , ^, ~, <<, >>

Overloading complex: +

```
#ifndef _COMPLEX_H_
#define _COMPLEX_H
class complex {
        public:
                // constructor with default values
                complex(double a = 0, double b = 0);
                // accessors
                double  get_real() const;
                double  get_img() const;

                // mutators
                void set_real(double real)
                void set_img(double img);
                // math operators
                complex operator+(const complex & c) const;
        private:
                double       real_part; // the real part
                double       img_part; // the imaginary part
}
#endif
```

interface

Overloading complex: +

```
complex complex::operator+(const complex & c) const{
     double real, img;

     real = real_part + c.get_real();
     img = img_part + c.get_img();
     return complex(real, img);

}
```

In your program you can have:

```
complex x (4,5), y (5);
complex z;
z = x + y;  // just like built-in numbers
```

Destructor

- Destructor is a method used to deallocate are resources (I.e. pointers) that are associated with an object.
- The destructor is called and executed when:
  - The object goes out of scope
    - a local variable in a function
  - Operator delete is called on a pointer to an object.
- If your class allocates memory, files or any other resources as part of the constructor or mutator, then you must have a destructor that deallocates all of these.

N-dimensional vector class:

```
class n_vector {
    public:
          n_vector();
          n_vector(double points[], int dims);
          ~n_vector(); // destructor
          // accessors
          int get_dims() const;
          int get_coord() const;
    private:
          double *coords;
          int num_dims;
}
```

N-dimensional vector: constructor

```
n_vector::n_vector(double points[], int dims){
    if ((dims <1) || (points != NULL)){
        // error, throw exception
    }
    num_dims = dims;
    coords = new double [num_dims];
    For (int i=0; i < num_dims; ++i){
        coords[i]  = points[i];
    }
}
```

N-dimensional vector: destructor

```
n_vector::~n_vector(){
              delete [] coords;
              num_dims = 0;
}
```

Deallocates the array of coordinates!

```
void do_something(double pts[], int n) {
       n_vector v (pts, n);
       …
        // do something with vector v
       …
       // when the end of function is reached
       // the destructor of v will be called to
       // free the memory
}
```

Copy constructor

- Mechanism to initialize a new object from an existing one.
- C++ automatically gives you a default one:
  - Makes a shallow copy of the objects.
    - Copies the basic type by value
    - Copies the pointers by simply copying the address value
    - Copies the object by calling their copy constructor
- Used:
  - Explicitly – ex: complex w (1, 2), u(w);
  - Implicitly – to create temporary objects in call by value
- Note: If you want independent objects (no shared references nor pointers in data member) then you must implement a copy constructor.

N-dimen. vector: copy constructor

```
n_vector::n_vector(const n_vector& v){
     num_dims = v.get_dims();
     coords = new double [num_dims];
     for (int i=0; i < num_dims; ++i){
          coords[i]  = v.get_coord(i);
     }
}
```

This process is called a deep copy of the object

Copy assignment operator

- Use to copy one object to another, when both objects have already been created.
- Behavior should be the same as that of the copy constructor.
- Example:

```
n_vector v (ptrs, n), z;
… // some  code does stuff here.
z = v ; // have assigned v to z
```

Copy assignment operator

```
const n_vector& n_vector::operator=(const n_vector& v){
    num_dims = v.get_dims();
    coords = new double [num_dims];
    for (int i=0; i < num_dims; ++i){
        coords[i]  = v.get_coord(i);
    }
}
```