UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 03:

Classes (I) – Basics, Constructors and Destructors

Learning Objectives

You should be able to review/understand:

- Procedural vs. object-oriented programming
- C++ structures versus C structures
- Classes
  - –Accessing class members
  - –Member functions
  - –Allocating objects dynamically
- Constructor and destructor functions
  - –Constructors: including copy constructors
  - –Destructors
- Composition

Procedural vs. Object-Oriented

- C++ supports procedural programming which
  - –solves a variety of engineering problems
  - –decreasingly efficient for large and complex program development
- C++ also supports object-oriented programming which
  - –is more natural as the logic applied to real-life problems
  - –programmers have difficulty in adopting such paradigm

Procedural Programming's Problem

- Procedural programming paradigm focuses on program's functionality
  - –How to represent data is not concerned
  - –`func_1` + `func_2` + … + `func_n`
- For large and complex programs, procedural programming faces the difficulties of
  - –maintaining and modifying the program
  - –debugging and following its logic
  - –too many disorganized, overloaded details
  - –creation of inadvertent logic errors

Sample Problem from Structure

```
struct SScore {
    char name[20];
    int  subj[3];
};
double computeAverage(SScore one) {
    return (one.subj[0]+one.subj[1]+
            one.subj[2])/3;
}
```

- Potential problems:
  - –the number of subjects is changed to `4`
  - –`computeAverage()` may change the user name unintentionally

OOP Paradigm

- OOP overcomes the above problems by
  - –using a collection of *objects* that communicate with each other through their interface functions
  - –focusing on both *data* and *operations*
- Three important concepts in OOP:
  - –*Encapsulation* binds data and functions into one capsule (object)
  - –*Inheritance* enables new codes to be derived from existing codes
  - –*Polymorphism* uses the same functions on different types of objects

Concept of Class

- *Class* is a foundation of OOP
  - –expands structure by binding together data and functions
  - –variables in class types are *objects*
- A object has its own unique identity, state and behaviors
  - –*states* ⇨ data fields
  - –*behaviors* ⇨ a set of functions
- *Class is the abstraction of objects* ⇔ *A object is an instance of class*
  - –Class is abstract ⇨ takes no memory
  - –Object is concrete ⇨ takes memory space

Class Declaration (1/2)

- Defined similar to structures

```
class class_name
{
public:
    //public variables and functions
protected:
    //protected variables and functions
private:
    //private variables and functions
};
```

class head

member access modifier

class body

- –Class name must be a legal identifier, typically starting with `C` or `T`
- –Class body includes many *data members* (variables) and *member functions* (methods)

Class Declaration (2/2)

- *Member access modifiers* can appear in an arbitrary order or multiple times
  - `public`: members can be accessed by members of its class or those of any other class or any non-member function
  - `private`: members serve only as utility functions for others of *the same class*
  - `protected`: used only with inheritance; members can be accessed by other members of *its class* or *the derived* class
- The default modifier for `class` is `private` ⇔ the default for `struct` is `public`

Two Ways to Define Classes

- First declare class, then define objects ⇨ the most common

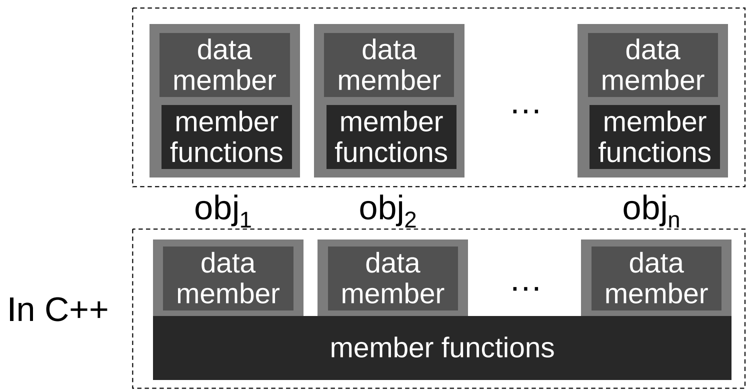
```
class someclass
{
    ...//implement data and functions
};
someclass obj1, obj2, ..., objn;
```

- Declare class and define objects right away

```
class someclass
{
    ...//implement data and functions
} obj1, obj2, ..., objn;
```


Memory Allocation for Objects

- Need to allocate memory for data members and member functions of every object
  - Objects of the same class use the same functions ⇨ memory waste



Access Members in Objects

- Members in objects can be accessed by
  - (1) object name and dot (`.`) operator

```
CScore stu1;
cout << stu1.name;
stu1.computeAverage();
```

  - (2) a pointer to the object and arrow (`->`) operator

```
CScore * pStu = &stu1;
pStu->computeAverage();
```

  - (3) a reference to the object and dot (`.`) operator

```
CScore & rStu = stu1;
rStu.computeAverage();
```

Scopes for Class (1/2)

- The scope of a class is enclosed by { and }
  - –can define variables and functions
  - –Variables in class cannot be modified by `auto`, `register` and `extern` (except `static`)
  - –Functions in class cannot be modified by `extern` but `inline`
- Scope for class name typically starts from *the declaration line* to *the file end*
  - –If put in the header file, its scope starts after the preprocessor directives to the end of the program

Scopes for Class (2/2)

- Class prototype scope: declares the class name *before being used*

```
class ⟨name⟩;
```

- Example

```
class COne;  //class prototype
class COne;  //repetition is OK!
class CTwo {
    ...
    COne a;  //use declared class
};
class COne { //the actual body
   ...
};
```


Functions in Class

- *Member functions* are one kind of functions
  - –Class without member functions = struct
  - –Belongs to class ⇨ need to consider *accessibility* (access modifier)
- Member functions can be defined into
  - –Functions defined in the class body: default as inline functions that are allowed to be included in the header files
  - –Function declared in the class body: the typical case; function definitions are written outside the class

Member Functions (1/3)

- Example for member functions in class body

```
class CScore
{
private:
    char name[20];
    int  subj[3];
public:
    double computeAverage() {
        return (subj[0]+subj[1]+subj[2])/3.0;
    }
    void setName(char * inName) {
        strncpy(name, inName, 20);
    }
    ...
};
```

Member Functions (2/3)

- Member functions outside class body

```
class CScore
{
private:
    char na
    int  su
public:
    double computeAverage();
    void setName(char * inName);
    ...
};
inline double CScore::computeAverage()
{
    return (subj[0]+subj[1]+subj[2])/3.0;
}
```

unlike normal functions, parameters in member functions need to be specified

default is not `inline` ⇨ need to specify

field qualifier (`::`)
⇨ tell the function belongs to what class

Member Functions (3/3)

- Separating declaration from implementation is due to (1) *information hiding* and (2) *intellectual property protection*
- You are free to change the implementation but the client program needs not to change as long as the declaration is the same

```
double CScore::computeAverage()
{
    return ((1*subj[0]+2*subj[1]+
3*subj[2])/3.0); //weighted version
}
```

- As a software vendor, only provide the customer with the header file and class object code without revealing the source codes

Preventing Multiple Declarations

- A common compiling error is to include the same header *files multiple* times in a program
  - Ex: `head1.h` includes `circle.h` and `testHead.cpp` includes `head1.h` and `circle.h`
- *Preprocessor directives* solve this issue

```
#ifndef CIRCLE_H
#define CIRCLE_H

//original class declaration in circle.h
class CCircle {
    ...
};

#endif
```


Accessor and Mutator Functions

- The private data field cannot be accessed outside the class.
  - to make them readable, provide a *get* function to return the field's value ⇨ *accessor* functions

```
returnType getPropertyName()
```

  - to make them updatable, provide a *set* function to set a new value in the field ⇨ *mutator* functions

```
void setPropertyName(datatype value)
```

Example of Accessor & Mutator

```
class CScore
{
private:
    char name[20];
    int  subj[3];
public:
    double computeAverage() {
        return (subj[0]+subj[1]+subj[2])/3.0;
    }
    void setName(char * inName) { //mutator
        strncpy(name, inName, 20);
    }
    char *getName() { //accessor
        return name;
    }
};
```


Constructors and Destructors

- Many errors starts from incorrect initialization or clearance of variables in C++
  - two special member functions: *constructors* and *destructors*
- *Constructors* aims at *assigning* values for data members when creating objects
  - ⇨ *object initialization*
- *Destructors* aims at *freeing* memory space for data members when destroying objects
  - ⇨ *object cleaning*

Constructor Functions (1/2)

- Typically, can only *access private* data member by calling member functions manually
  - Constructor functions *automatically runs* when creating an object

```
class CScore
{
private:
    char name[20];
    int  subj[3];
};
```

```
CScore one = {“Tom”, {66, 70, 80}};
```

```
CScore one;
one.setValue(“Tom”, {66, 70, 80});
```


Constructor Functions (2/2)

- A constructor function has the name as the class itself with or without parameters
  - can be *overloaded* ⇨ multiple functions with the same names
  - can have *default* parameter values
- May call different versions of constructors

```
CScore one;
one.CScore(“Tom”, {66, 70, 80});
one.CScore(“Tom”);
one.CScore({66, 70, 80});
one.CScore();
```

  - imply overloaded functions

Example of Constructor Versions

```
CScore::CScore(char* str, double* dary) {
    strncpy(name, str, 20);
    subj[0] = dary[0];
    subj[1] = dary[1];
    subj[2] = dary[2];
}
```

```
CScore::CScore(char* str) {
    strncpy(name, str, 20);
}
```

```
CScore::CScore(double* dary) {
    subj[0] = dary[0];
    subj[1] = dary[1];
    subj[2] = dary[2];
}
```

```
CScore::CScore() {;}
```


More about Constructors

- Constructors are like other normal member functions but
  - –have the same name as the class
  - –cannot specify the return type, even void
  - –must be *public* (or can be *protected* for derived classes)
  - –*avoid ambiguity* from the default parameters in overloaded functions

```
CScore::CScore(char* str,
               double* dary={0,0,0}) {...}
```

```
CScore::CScore(char* str) {...}
```

```
one.CScore(“Tom”); //call which version?
```


Default Constructor

- If no constructor is defined, the compiler implicitly generate a *default constructor* without any parameter
  - –Ex: `CScore::CScore() {;}`
- If any constructor is defined, need to specify a default constructor *explicitly*

```
CScore two;//error if no default const.
```

- Initialized values in default constructor depend on the datatype of the object
  - –initialized values are random, ex:

```
CScore one;
```

  - –initialized values are *0* or *empty*, ex:

```
static CScore one;
```


Copy Constructor

- What if using a existing object to initialize the new object? Ex:

```
CScore one(old);//old is declared before
```

  - –need a different type of constructors
- *Copy constructor*

```
〈CNAME〉::〈CNAME〉(const 〈CNAME〉 & 〈var〉) {
    //copy constructor: function body
};
```

  - –`〈CNAME〉` is the class name
  - –`〈var〉` is the name for formal parameter of copied object
  - –If no copy constructor is specified, the compiler *automatically* generate one

Example of Copy Constructors (1/2)

```
class CStr
{
private:
    char * line;//default access is private
public:
    CStr(char* word); //A
    CStr(const CStr & old); //B
    void ShowCStr();
};
```

```
int main() {
    CStr one(“prog3-1”); //call A
    CStr two(one); //call B; CStr two = one;
    two.ShowCStr();
    return 0;
}
```


Example of Copy Constructors (2/2)

```
CStr::CStr(char * word) //A
{
    line = new char [strlen(word)+1];
    strcpy(line, word);
}

CStr::CStr(const CStr & old) //B
{
    line = new char [strlen(old.line)+1];
    strcpy(line, old.line);
}

void CStr::ShowCStr(){
    cout << line << endl;
}
```


Destructor Functions

- *Destructor function* is the complement of a constructor function
  - need to *clean up* the object
  - unlike constructors, *only one destructor*
  - the compiler *automatically generates* one if no destructor is declared
- A destructor has the following properties:
  - its name = tilde (~) + class name
  - should be public
  - cannot have a return type
  - cannot have any parameter
  - *automatically called* when the object goes out of scope

Example of Constructor/Destructor (1/2)

```
class CStr
{
private:
    char * line;
public:
    CStr() { line = NULL; } //A
    CStr(char* cline) { line = cline; } //B
    ~CStr() {}
};
```

```
int main() {
    char* p = “prog3-2”;
    CStr one(p); //call B

    one.~CStr(); //call destructor
    return 0;
}
```

Example of Constructor/Destructor (2/2)

- Modified constructor and destructor

```
class CStr
{
private:
    char * line;
public:
    CStr() { line = NULL; } //A
    CStr(char* cline) {
        line = new char [strlen(cline)+1];
        strcpy(line, cline);
    } //B
    ~CStr() {
        if (line) delete [] line;
        line = NULL; cout << “done” << endl;
    }
};
```


Understanding Composition

- *Composition* ⇨ uses an object of one class within the object of another class
  - –forms a “has-a” relationship
  - –top-level class are called *composed classes*
  - –contained objects are called *member objects*
- Example: A `CScore` class has a `CStr` object

```
class CScore //composed class
{
private:
    CStr name; //member object
    int  subj[3];
public: ...
};
```


Example for Composed Class (1/3)

```
class CPoint {
    int x, y;
public:
    CPoint() { x=0; y=0; }
    CPoint(int a, int b) { x=a; y=b; }
    void set(int a, int b) { x=a; y=b; }
    void move(int a, int b) { x+=a; y+=b; }
};
```

```
class CRect {
    CPoint p1, p2;//give two points
public:
    CRect() { p1.set(0,0); p2.set(0,0); }
    CRect(int a, int b, int c, int d) {
        p1.set(a,b); p2.set(c,d); }
    CRect(const CPoint &q1, const CPoint &q2){
        p1 = q1; p2 = q2; }
};
```


Example for Composed Class (2/3)

- Can `CRect` be declared as follows?

```
class CRect {
    CPoint p1(0,0); //top-left point
    CPoint p2(0,0); //bottom-right point
    ...
};
```

- Can a `CRect` constructor be defined as follows?

```
CRect::CRect(int a, int b, int c, int d) {
    p1(a,b); //top-left point
    p2(c,d); //bottom-right point
    ...
}
```

- Both answers are NO!

Example for Composed Class (3/3)

- The first solution for legal initialization is

```
   CRect(int a, int b, int c, int d) {
       p1.set(a,b); p2.set(c,d); }
   CRect(const CPoint &q1, const CPoint &q2){
       p1 = q1; p2 = q2; }
```

- The second solution creates a *new copy constructor* in class `CPoint`

```
//class CPoint
   CPoint(int a, int b) { x=a; y=b; }
   CPoint(const CPoint &old) {
       x = old.x; y = old.y; }
```

```
//class CRect
   CRect(int a, int b, int c, int d) :
       p1(a,b), p2(c,d) {}
   CRect(const CPoint &q1, const CPoint &q2):
       p1(q1), p2(q2) {}
```


Summary (1/2)

- Review OO concept and programming:
  - Procedural programming's problem
  - Three concepts of OOP paradigm
  - What is the relationship between class and object?
- Class declaration
  - Data members and function members
  - Three access modifiers
  - Two ways to define classes and declare objects
  - Memory allocation for objects

Summary (2/2)

- Using objects
  - Three ways to access members in objects
  - Scopes for class
- Member functions
  - Separate declaration from implementation
  - Preventing multiple declarations
  - Accessor and mutator functions
  - Constructors: default and copy ones
  - Destructors
- Composition
  - Composed classes and object members
  - Legal initialization