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

Lecture 04:

Classes (II) – Advanced Topics

Learning Objectives

You should be able to understand:

- Object and pointer
  - dynamical memory for objects
  - pointers to methods and members
  - this pointer
- Different kinds of data members and methods
  - const data and methods
  - mutable and explicit members
- Advanced composition
  - Nested classes
- Friend function and class

Objects and Pointers

- C++ expands dynamic allocation to objects
  - allow `malloc()`/`free()` and `new`/`delete`
  - `malloc()` does not call the constructor when an object is created
  - Use `new`/`delete` for objects
- Heap objects
  - Format:

```
⟨CName⟩ * ⟨pObj⟩ = new ⟨CName⟩;
delete ⟨pObj⟩;
```

  - Ex:

```
CScore * pStu = new CScore;
pStu->computeAverage();
delete pStu;
```


Object Array

- Similar to `struct` arrays, class can be used to create object arrays
  - format: `⟨CName⟩ ⟨pObj⟩[iSize];`
  - use public member: `⟨pObj⟩[iIdx].member;`
- Example: `lec4-1.cpp (part 1)`

```
CScore StuAry[3];
//CScore[0]=("Adam",{43,62,85});
//CScore[0]=("Bill",{72,82,93});
//CScore[0]=("Nick",{31,53,76})};
double sum = 0;
for (int i=0; i<3; i++) {
    sum += StuAry[i].computeAverage();
}
cout << sum/3.0 << endl;
```

Pointer to Member Function (1/2)

- Pointer to normal functions (in Lecture 02)
  - –Format: `⟨rtn_type⟩ (*⟨pFun⟩)(⟨para_list⟩);`
  - –Ex:

```
double (*pF)();
...
pF = Fun;
double result = (*pF)();
```

- What if ? `pF = &StuAry[0].computeAverage;`
  - –compilation error!
  - –because the return type of the pointer should match the type of member function
  - –`double` is not equal to `CScore::double`

Pointer to Member Function (2/2)

- Pointer to member functions

```
⟨datatype⟩ (⟨CName⟩ ::*⟨pFun⟩)(para_list);
```

  - –Ex:

```
double (CScore::* pF)();
...
pF = &CScore::computeAverage;
```

- Use the pointer to the member function, Ex:

```
//lec4-1.cpp (part 2)
CScore * p1 = &(StuAry[2]);
cout << p1->computeAverage();
double (CScore::* p2)() =
        &CScore::computeAverage;
cout << (p1->*p2)();
```

  - –member functions are not put at the space of objects but that all class objects share

Pointer to Data Member

- Similar to the pointer to member function, the pointer to data member has the format

```
⟨datatype⟩ (⟨CName⟩::* ⟨pName⟩);
⟨pName⟩ = &⟨CName⟩::⟨data_member⟩;
```

- Ex: see more in `lec4-1.cpp(part 3)`

```
class X {
public:
    int a;
    void f(int b) { cout << b << endl;}
};
```

```
int X::* pa = &X::a;
X obj1;
obj1.*pa = 10;
cout << “value a = ” << obj1.*pa;
```


`this` Pointer (1/3)

- How C++ guarantee that the data member is correctly referenced via member function?
- Example:

```
class Box {
public:
    Box(int h=6, int w=8, int l=10):
        hgt(h), wid(w), len(l) {}
    int vol() { return hgt*wid*len; }
private:
    int hgt, wid, len;
};
```

```
Box a(2,4,6), b(3,5,7);
```

  - –`a.vol()` and `b.vol()` use the same function

`this` Pointer (2/3)

- Each member function contains a special pointer named **`this`**
  - points to the address of the called object

```
(this->hgt)*(this->wid)*(this->len);
```

  - If call `a.vol()`, equivalently,

```
(a->hgt)*(a->wid)*(a->len);
```

  - `this` pointer is used implicitly but works like

```
//in Box class declaration, version 2
int vol(Box * this) {
    return (this->hgt)*(this->wid)
*(this->len); }
```

and call by

```
a.vol(&a);
```


`this` Pointer (3/3)

- Explicitly use `this` pointer if needed

```
//in Box class declaration, version 1
int vol() {
    return ((*this).hgt)*((*this).wid)
*((*this).len); }
```

  - pair of parentheses cannot be omitted because dot (`.`) operator precedes dereference (`*`) operator
  - If **`*this.hgt`** is interpreted as **`*(this.hgt)`**
    ⇨ but this is a pointer and `this.hgt` cannot be parsed correctly
    ⇨ Compilation error! ⇨ use **`this->hgt`**

Example of `this` Pointers (1/2)

```
class CPoint {
    int x, y;
public:
    //this is hidden in setPt() and Print()
    void setPt(int c, int d) { x=c; y=d; }
    void Print() { cout << x << " "
                        << y << endl; }
};
```

```
class CPoint {
    int x, y;
public:
    //use this to refer the current object
    void setPt(int x, int y) {
        this->x=x; this->y=y; }
    void Print() { cout << this->x << " "
                        << this->y << endl; }
};
```


Example of `this` Pointers (2/2)

```
class CPoint {
public:
    CPoint offset(int diff) {
        x += diff; y += diff;
        return *this; //real use of this
    }
};
```

```
//in main()
CPoint p, q;
int a=3, b=5;
p.setPt(a,b);
p.Print();
q = p.offset(3);
q.Print();
p.Print();
```

What are going to be printed on screen?

Summarize `this` Pointer

- A `this` pointer stores the *address of an object* used to call a non-static member function
  - typically *hidden from programmer*
  - *handled by the compiler*
  - address of that object is passed to the function and stored in `this`
  - access the data stored in the object by dereferencing `this`, i.e. `this->[member]`
  - use when returning the object itself or when parameters have the same name as private data members
- ⇨ be aware of `this` for advanced OOP

Different Kinds of Members

- C++ has many choices of members
  - `static` members: members that all objects of that class share
  - `const` members: members that require initialization and cannot be updated
  - `mutable` members : members that need no protection from changing any time but are encapsulated in class
  - `explicit` members: members that avoid ambiguity from calling constructor implicitly
  - and more (reference members)

Static Data Members

- Class provides data encapsulation and hiding
  - but results in difficulties data sharing and external visit
  - can be resolved by global variables
- A better solution ⇨ static members
  - is specific to one class
  - has a scope shared by the objects of the same class
- A static member can be accessed
  (1) with class methods or
  (2) outside class methods

  `⟨type⟩⟨class⟩::⟨static_data⟩ = ⟨value⟩;`

Define Static Data Members

- A static data member is defined by
  - use keyword `static` to declare a data member in class
  - allocate memory and initialize outside class
  - not limited by the access modifier
- Example

```
class X {
private:
    //declare a static data variable
    static int count;
};
//mtd(1): initialize the static member
int X::count = 0;
```

Access with Class Methods

- Use static data members from class methods

```
class CNum {
public:
    CNum(int a) { x = a; y += x; }
    static void fun(CNum m) {
        cout << m.x << "vs." << y << endl; }
private:
    int x;
    static int y;
};
int CNum::y = 0;
```

```
//in main()
CNum O1(4), O2(7);
CNum::fun(O1);
CNum::fun(O2);
```

What to display on screen?

Use Static Members (1/3)

- Design a `CScore` class with data members: id number (`id`), name (`name`), three subject scores (`subj[3]`) , and total sum of each subject scores (`sum[3]`). Write functions to compute the sums of each subjects.

```
//lec4-3.cpp (part 1)
class CScore //data members in CScore
{
private:
    char id[6];
    char name[20];
    int  subj[3];
    static int sum[3];
}
```


Use Static Members (2/3)

```
//member functions for CScore
public:
    CScore(char *, char *, int *);
    void showSCore() {
        cout << id << " " << name << " "
        << subj[0] << " " << subj[1] << " "
        << subj[2] << endl;
    }
    static void showSum() {
        cout << sum[0] << " " << sum[1] <<
        " " << sum[2] << endl;
    }
};
int CScore::sum[3] = {0,0,0};
//or int CScore::sum[3];
```


Use Static Members (3/3)

```
//Constructor for CScore objects
CScore::CScore(char *uid, char *uname,
                int *uscore) {
    subj[0] = uscore[0];
    subj[1] = uscore[1];
    subj[2] = uscore[2];
    strcpy(id, uid); strcpy(name, uname);
    sum[0] += subj[0];
    sum[1] += subj[1];
    sum[2] += subj[2];
}
```

```
//in main(){}
//CScore S1("99123", "Tom", {70, 80, 90});
S1.showSCore(); S1.showSum();
//CScore S2("99145", "Bill", {60, 40, 50});
S2.showSCore(); S2.showSum();
```

const Members (1/2)

- Constants almost never make sense at the object level ⇨ often used with `static` modifier

```
class CScore { //lec4-3.cpp (part 2)
protected:
    //declare a static constant
    static const int Max;
};
const int CScore::Max = 100; //outside class
```

```
//modified constructor in CScore
CScore::CScore(char *uid, char *uname,
                    int *uscore) {
    subj[0] = (uscore[0]>Max)?Max:uscore[0];
    subj[1] = (uscore[1]>Max)?Max:uscore[1];
    subj[2] = (uscore[2]>Max)?Max:uscore[2];
...}
```


const Members (2/2)

- `const` can used with `&` modifier ⇨ need to be initialized in constructors

```
class CScore { //lec4-3.cpp (part 2)
public:
    //declare a constant reference
    const int & sc;
};
```

```
//modified constructor in CScore
CScore::CScore(char *uid, char *uname,
    int *uscore) : sc(subj[0])    {
    subj[0] = (uscore[0]>Max)?Max:uscore[0];
    subj[1] = (uscore[1]>Max)?Max:uscore[1];
    subj[2] = (uscore[2]>Max)?Max:uscore[2];
...}
```


const Methods

- Only constant methods can call a `const` object or a reference to `const` object

```
class CScore {
public: //add more const methods
    char* getName() const { return name; }
    void setName(const char* uname) {
        strcpy(name, uname);} //not const
};
```

```
//CScore S1("99123", "Tom", {70, 80, 90});
cout << S1.getName() << endl; //OK!
S1.setName("Tom Mitchell"); //OK!
const CScore & S3 = S1;
cout << S3.getName() << endl; //OK!
S3.setName("Bob"); //compilation error!
```


mutable Data Members

- `mutable` means volatile ⇔ constant
  - a constant function cannot modify any data member which may not need protection
- `mutable` makes data member always changeable, even in a `const` method
  - cannot modify methods but regular variables
  - can modify `const` and `static`, not reference

```
class CScore {//lec4-3.cpp (part 3)
public:
    mutable int sCount = 0;           };
```

```
void CScore::showName() const {
    cout<<sCount++<<":"<<name<<endl;   }
```

More on `mutable` Members

- Modify `mutable` data by constant objects

```
class COne {
public:
    mutable int x; int y;                };
```

```
const COne m;
m.x = 10; //legally used
m.y = 20; //not legally used
```

- `const` + `mutable`: leave for self-exploration

```
class CTwo {
    //good: ptr to a mutable constant
    mutable const int* p;
    //bad: a mutable constant pointer
    mutable int* const q;                };
```


`explicit` Members

- Two types of calling constructors:
  - explicit call `⟨CName⟩ ⟨Obj⟩(⟨init_values⟩);`
  - implicit call `⟨CName⟩ ⟨Obj⟩ = ⟨init_values⟩;`
- Distinguish implicit call and object assignment?

```
CScore::CScore(const char* cstr) {
    cout << cstr << endl;
}
```

```
CScore one = "Tom"; //assign or construct?
```

- `explicit` is used to modify constructors of a class and use them explicitly

```
explicit CScore::CScore(const char* cstr)
{ ... }
```

```
CScore one = "Tom"; //illegal now!!
```


Advanced Composition

- **Nested class** : a class `CTwo` is declared within the scope of another class `COne`
  - used when the inner class is meaningful inside the outer class
  - if `CTwo` is `public`, use with `COne::`
  - format:

```
class COne { //outer class
    ...
    class CTwo { //inner class
        ...
    };
};
```

  - Nested class cannot access any `private` member of the outer class by default

Example of Nested Class

```
class Node { ... }; //outer class
class Graph {
public:
    //Graph::Node hides ::Node in Graph's {}
    class Node { ... }; //inner class
    //resolves to nested Graph::Node
    Node *grpah;
};
//Graph::Node is not visible globally
Node *pnode;
class Chain {
public:
    //Chain::Node hides ::Node in Chain's {}
    class Node { ... }; //inner class
    //resolves to nested Chain::Node
    Node *chain;
};
```

More on Nested Classes

- A nested class `CTwo` in `COne` can have the same kind of members as a *nonnested* class
  - Example:

```
class Chain { //outer class
public:
    class Item { //all members are private
        friend class Chain;
        Item(int val = 0); //constructor
        Item *next; //point to its own class
        int val;
    };
    //...
private:
    Item *chain;
    Item *at_end;
};
```


Defined Outside Outer Class

- A nested class can also be defined outside the enclosing class
  - need to take care of *qualifier*
  - only *pointers* and *references* to the inner class can be declared

```
class COne { //enclosing class
    class CTwo; //nested class
    CTwo *pobj; //CTwo obj is wrong!!!
    //...
};
class COne::CTwo {
    CTwo(int val=0);
    CTwo *next;
    int value;
};
```


Later Definition of Inner Class

- A nested class can be first declared and then later defined in the outer class

```
class Chain { //outer class
private:
    class Item; //declare Chain::Item
    class Ref {
        Item *pIt; //has type Chain::Item*
    };
    class Item { //later define Chain::Item
        Ref *pref; //has type Chain::Ref*
    };
    //...
};
```


`friend` Functions

- Hiding data inside a class and letting only class member functions have direct access to private data is a very important OOP concept
- But C++ also provides another of function to access members in class ⇨ `friend` functions
  - `friend` functions are *not member* functions but *can still access* class `private` members
  - *defined outside* of class scope
- Reasons to have `friend` functions:
  - to access private members of two or more different classes
  - for I/O or operator (in Lecture 6) functions

Properties of `friend` Functions

- Properties:
  - placed *inside the class definition* and preceded by the `friend` keyword
  - *defined outside the class* as a normal, non-member function
  - called like a normal non-member function.
- If a function is a friend of two or more different classes, each class *must contain* the friend function *prototype* within its body
- A friend function cannot be *inherited* (covered in Lecture #7) but can be a *member* of one class

Example of `friend` Functions

```
//lec4-4.cpp
class CPoint {
    int x, y;
    friend CPoint offset(CPoint &, int);
public:
    CPoint() { x=0; y=0; }
    CPoint(int a, int b) { x=a; y=b; }
    void Print() { cout << x << " "
                        << y << endl; }
};
CPoint offset(CPoint &pt, int diff) {
    pt.x += diff; pt.y += diff;
    return pt;
}
```

```
//in main()
CPoint p1( 3, 5 ); p1.Print();
offset(p1, 4); p1.Print();
```


`friend` Classes

- An entire class can be a friend of another class ⇨ `friend` class
  - can be used when all member functions of one class should access the data of *another class*
  - require *prototypes* to be placed within each class *individually*
- An entire class can be designed as `friend`
  - all its member functions automatically granted a friendship by the class
  - but "class `B` is a `friend` of class `A`" does not imply "class `A` is a `friend` of class `B`"

Example of `friend` Class (1/3)

```
//in CPoint.h
class CPoint {
    friend class CLine;
    int x, y;
    void Offset(int diff) {
        x += diff; y += diff;
    }
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 Print() {
        cout << x << " " << y << endl;
    }
};
```

Example of `friend` Class (2/3)

```
//in CLine.h
class CLine {
    CPoint p1, p2;
public:
    CLine(int x, int y, int w, int z) {
        p1.x = x; p1.y = y; //access private
        p2.x = w; p2.y = z;           }
    void Print()() {
        //call public Print in CPoint
        cout << “Point 1:”; p1.Print();
        cout << “Point 2:”; p2.Print(); }
    void Display() {
        offset(p1,100); //call friend func
        p2.Offset(200); //call private func
        Print(p1, p2); }
};
```


Example of `friend` Class (3/3)

```
//in main()
#include “CPoint.h”
#include “CLine.h”

int main()
{
    CPoint p1(2,4); p2(6,8);
    p1.Print(); p2.Print();
    p1.Offset(3); //error! Why?

    CLine l1(1,3,5,7), l2(2,4,6,8);
    l1.Print();
    l2.Display();

    return 0;
}
```


Summary (1/2)

- Review object and pointers with class:
  - What is object array?
  - How to use pointers to members in class?
  - this pointers
- Different kinds of Members
  - What is a static data member?
  - Why do we need const methods?
  - What is a mutable data member? Why do we need mutable
  - What is a explicit members? When to apply explicit?

Summary (2/2)

- Advanced composition
  - What is a nested class and a enclosing class?
  - Can a nested class include a member of the same kind?
  - How to define the nested class outside the enclosing class?
  - How to access the public and private member of outer class?
- friend
  - When to use a friend function?
  - When to use a friend class and how?