Programming 2

Object Oriented Programming

Daniel POP

Unit 4

Agenda

1. Self-reference
2. Modifiers
    1. static
    2. const
    3. mutable
    4. friend
3. Nested classes
4. Concrete classes
5. Plain old data objects

Modifiers: static (I)

DEFINITON [static data member] A variable that is part of a class, yet is not part of an object of that class is called a static member.

- There is exactly one copy of a static variable 'shared' by all objects of that class.
- Static data member = 'global variable' defined and accessible only in the scope of that class.

DEFINITION [static member function] A function that need access to static members of a class, yet doesn't need to be invoked for a particular object, is called a static member function.

- Static member function = 'global function' defined and accessible only in the scope of that class.

In order to clearly differentiate between static and non-static members + enhance code readability, the first (bold) syntax is recommended to be used.

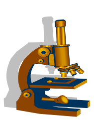
Syntax

```
// declaration
static <member decl.>;

// access
X::memberName;
Obj.memberName;
```

Modifiers: static (II)

- *static data members have to be defined/initialized somewhere (in the class definition they are **only declared!**)*
- static member functions **DO NOT** receive this pointer
- => the access to non-static members (data/function) is not possible from within static functions
- creation, initialization and access to static members are independent of objects existence.
- static members are prone to race conditions in multi-threaded code!
- Example: default date (today) in Date class



Memory representation of static data.

```
class Date {
     int day, month, year;
     static Date today; // declare the static data member
public:
     // …
     static void initToday();
};

Date Date::today; // create the static data member

void Date::initToday() {
     time_t t;
     time(&t);
     tm* now = localtime(&t);
     day = now->tm_mday; // ERROR: whose day is this day?
     today.day = now->tm_mday;
     today.month = 1 + now->tm_mon;
     today.year  = 1900 + now->tm_year;
}

int main(int, char*[]) {
     Date::initToday();
     return 0;
}
```

Self-Reference

```
void f() {
     Date d;
     d.init(25,12,2007).getMonth();
}
```

=> The prototype of init function should be:

Date& init(int, int, int);

```
Date& Date::init(int day, int month, int year) {
   this->_day = day; // ⇔ _day = day
   _month = month; // implicit use of this
   this->year = year; // year - makes possible
                      // to use the same name
                      // for members and args

   this++; // ILLEGAL: this is a const pointer
   return *this;
}
```

Each **non-static member** function knows what object it was invoked for and can explicitly refers to it using **this** keyword.

this is a pointer to the object for which the function was called

- For non-const function its type is: X* const this;
- For const function its type is: const X* const this;

Similar in Java (this), Smalltalk (self) or Simula (THIS).

Modifiers: const (I)

Data members

- Cannot be modified!
- Useful to declare constants at class scope

Constant data members must be initialized at declaration or in the initialization list of each constructor of the class.

Constant data members are helpful to implement immutable objects.

Usually, it makes sense to have constant static members because constant values are usually shared between all the instances of a class (e.g. MAX_VIEWS).

```
class X {
     const int ci=17;
     static const int MAX_VIEWS; // declaration

public:
};

// initialization of static const members
const int X::MAX_VIEWS = 256;
```

Syntax

```
const <member declaration>;
```

Modifiers: const (II)

Member functions

- Cannot modify the state of the object (i.e. data members)
- Enhances the code’s clarity
- Prevents accidental updates
- When the function is implemented outside its class, const suffix is required (see getMonth() example)

```
class Date {
     int day, month, year;
     static Date today; // declare the static data member
public:
     // const, inline member function
     int getDay() const {
          day = 0 ; // ERROR: we’re in const function
          return day;
     }
     int getMonth() const;
     void setMonth(int);
};

int Date::getMonth() const {
          return month;
}

int main(int, char*[]) {
     Date d;
     cout << d.getDay() << d.getMonth() << d.getYear();
     return 0;
}
```

– Calling non-const member functions on const objects?

```
void f(Date& d, const Date& cd) {
   d.getMonth();    // ERR vs OK?
   cd.getMonth();   // ERR vs OK?
   d.setMonth(3);   // ERR vs OK?
   cd.setMonth(3); // ERR vs OK?
}
```

Syntax

```
<member declaration> const;

<member declaration> const {
    // implementation
}
```

Modifiers: mutable

Applies only to **data members**

- Can always be modified, even in const functions!

- Useful for members that need to be changed in const functions and don't represent the "actual" internal state of the object

```
class Date {
     int day, month, year;
     mutable string cache; // always changeable!
     mutable boolean validCache; // always changeable!

public:
     // const member function
     string toString() const;
};

string Date::toString() const {
     if(!validCache) {
         cache = compute_new_string_representation();
         validCache = true;
     }
     return cache;
}
```

Syntax

```
mutable <declaration>;
```

Mutability through indirection

Alternative to mutable modifier is a technique named *Mutability through Indirection*, where the changing data is placed in a separate object (in example, Cache* c)

Cache* c – is not modified, rather the content it points to.

```
struct Cache {
    string cache;
    boolean validCache;
};

class Date {
     int day, month, year;
     Cache* c; // to be initialized in constructor(s)

public:
     // const member function
     string toString() const;
};

string Date::toString() const {
     if(!c->validCache) {
         c->cache = compute_new_string_representation();
         c->validCache = true;
     }
     return c->cache;
}
```

Friends (I)

```
class Matrix { /* declarations */ };

class Vector { /* declarations */ };

Define a function that: Vector x Matrix → Vector
```

- Rationale: functions need access to private members of one class
- Solution: make those functions members of the class
- What happens if one function need to have access to private members of 2 or more classes? It can be member of only one class :-(

- Friends – help us to do this :-)

Friends (II)

DEFINITION [Friend functions] Friend functions are functions that are not members of one class, yet they have access to private members (data + functions) declared in that class.

DEFINITION [Friend class] If class X is friend class of class Y, then all member functions of class X are friend functions (i.e. have access to private members) of class Y.

It's not relevant the access control modifier (private, public, protected) used to declare a friend function/class, because **friend functions are not member of the class they are friend of!**

Friends can be either functions that are member of another class, or global functions.

Syntax

```
friend ftype fname([arg_list]);

friend class X;
```

Friends (III)

```
class Matrix; // Forward declaration

class Vector  {
     float v[4];
public:
     Vector(float v0=0, float v1=0, float v2=0, float v3=0);
     float operator[] (int index);
     friend Vector multiply(const Vector&, const Matrix&);
};

class Matrix {
     Vector rows[4];
     friend Vector multiply(const Vector&, const Matrix&);
};
```

```
Vector multiply(const Vector& vec, const Matrix& mat) {
     cout << v[0]; // OK or ERROR
     cout << rows[0][0]; // OK or ERROR
     cout << vec.v[0]; // OK or ERROR
     cout << mat.rows[0][0]; // OK or ERROR
}

void main() {
      Matrix m;
      Vector v(1.0, 2.5, 5.0, 6.0);
      Vector w = multiply(v, m); // w = v x m
      cout << m.rows[0][0]; // OK or ERROR?
}
```

Friends (IV)

One of the key principle OO is data hiding (encapsulation), but sometimes is to restrictive and needs to be “broken”

Not recommendable to use friends; use only if it’s impossible to solve the problem otherwise.

Friends (V)

- More examples: see operator overloading
- Finding friends: declared before or identifiable by argument type
- Friendship **is NOT** reflexive nor transitive
- Friendship is not inherited

Action	**Non-static member function**	**Static member function**	**Friend function**
Has access to private members	x	x	x
Is declared in the class scope	x	x	
(transparently) Receives *this* pointer	x		

Member or Friend?

- Some operations can be performed only by members: constructor, destructor, virtual functions
- A function that has to access the members of a class should be a member unless there's a specific reason for it not to be a member.
- Prefer member functions over friends because of following OOP principles and a much clear syntax.
- Member functions must be invoked for objects of their class only; no user defined conversions are applied.
- For example, if transpose function transposes its calling object instead of creating a new transposed matrix then it should be a member of class.
- Example: (see also operator overloading for more examples)

```
class X {
public:
     X(int);
     void m1();
     friend int f1(X&);
     friend int f2(const X&);
     friend int f3(X);
};
```

```
void f() {
    7.m1();
    f1(7);


    f2(7);
    f3(7);
}
```

Member or Friend?

- Some operations can be performed only by members: constructor, destructor, virtual functions
- A function that has to access the members of a class should be a member unless there's a specific reason for it not to be a member.
- Prefer member functions over friends because of following OOP principles and a much clear syntax.
- Member functions must be invoked for objects of their class only; no user defined conversions are applied.
- For example, if transpose function transposes its calling object instead of creating a new transposed matrix then it should be a member of class.
- Example: (see also operator overloading for more examples)

```
class X {
public:
     X(int);
     void m1();
     friend int f1(X&);
     friend int f2(const X&);
     friend int f3(X);
};
```

```
void f() {
    7.m1(); // err: X(7).m1() is not tried!
    f1(7);  // err: f1(X(7)); is not tried because
               no implicit conversion is used for
               non-const references
    f2(7); // ok: f3(X(7));
    f3(7); // ok: f3(X(7));
}
```

In-class initialization and function definition

```
class Date {
     int day{today.day},     // in-class initialization
         month{today.month}, // in-class initialization
         year{today.year};   // in-class initialization

     static Date today;

public:
     int getDay() const {  // inline function
          return day;
     }
     int getMonth() const { // inline function
           return month;
     }
};

Date Date::today;
```

Member classes (nested classes)

```
class Tree {
    // member (nested) class
    class Node {
        Node* right;
        Node* left;
        int value;
    public:
        void test(Tree*);
    };

    Node* top;

    static Node* current;
public:
    void g(Node* p);
};

void Tree::Node::test(Tree* p) {
    top = right; // ERR: no object of type Tree specified
    p->top = right; // OK
    current = left; // OK: current is static in enclosing class
}

void Tree::g(Tree::Node* p) {
    int val = right->value; // ERR: no object of type Tree::Node
    int v = p->right->value; // ERR: Node::right is private
    p->test(this); // OK
}

Node* Tree::current = NULL; // ERR: Node not a type
Tree::Node* Tree::current = NULL; // OK
```

- No 'special' permissions for nested classes
- They need to be referred to using fully qualified names, i.e. prefixed by enclosing class name
- A nested class has access to private members declared in enclosing class (just as any member function), as long as is given an object of enclosing class

Concrete classes

DEFINITON [concrete class] A class is called concrete if its representation is part of its definition. This distinguishes it from *abstract* classes, which provide an interface to a variety of implementations.

```
enum class Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };

class Date {
public: // public interface:
    class Bad_date { }; // exception class
    explicit Date(int dd ={}, Month mm ={}, int yy ={}); // {} means ‘‘pick a default’’

     // non-modifying functions for examining the Date:
     int day() const;
     Month month() const;
     int year() const;
     string string_rep() const; // string representation
     void char_rep(char s[], int max) const; // C-style string representation

     // (modifying) functions for changing the Date:
     Date& add_year(int n); // add n years
     Date& add_month(int n); // add n months
     Date& add_day(int n); // add n days
private:
     bool is_valid(); // check if this Date represents a date
     int d, y; // representation
     Month m;
};

bool is_date(int d, Month m, int y); // true for valid date
bool is_leapyear(int y); // true if y is a leap year
bool operator==(const Date& a, const Date& b);
bool operator!=(const Date& a, const Date& b);
const Date& default_date(); // the default date
ostream& operator<<(ostream& os, const Date& d); // print d to os
istream& operator>>(istream& is, Date& d); // read Date from is into d
```

Concrete classes

We prefer concrete classes for small, frequently used, and performance-critical types, such as complex numbers, smart pointers or containers.

1. A constructor specifying how objects/variables of the type are to be initialized.
2. A set of functions allowing a user to examine a **Date** (namely accessors). These functions are marked **const** to indicate that they don't modify the state of the object/variable for which they are called.
3. A set of functions allowing the user to modify **Date**s without actually having to know the details of the representation or fiddle with the intricacies of the semantics (mutators).
4. Implicitly defined operations that allow **Date**s to be freely copied.
5. A class, **Bad_date**, to be used for reporting errors as exceptions.
6. A set of useful helper functions. The helper functions are not members and have no direct access to the representation of a **Date**.

“Plain Old Data” objects

DEFINITION: POD (“Plain Old Data”) is an object that can be manipulated as “just data” without worrying about complications of class layouts or user-defined semantics for construction, copy and move.

A POD object must
- not have a complicated layout (e.g. with a vptr)
- not have non-standard (user-defined) copy semantics, and
- have a trivial default constructor

```
struct S0 { }; // a POD
struct S1 { int a; }; // a POD

struct S2 { int a; S2(int aa) : a(aa) { } }; // not a POD (no default constructor)

struct S3 { int a; S3(int aa) : a(aa) { } S3() {} }; // a POD (user-defined default constructo
struct S4 { int a; S4(int aa) : a(aa) { } S4() = default; }; // a POD

struct S5 { virtual void f(); /* ... */ }; // not a POD (has a virtual function)

struct S6 : S1 { }; // a POD
struct S7 : S0 { int b; }; // a POD

struct S8 : S1 { int b; }; // not a POD (data in both S1 and S8)

struct S9 : S0, S1 {}; // a POD
```

Further Reading

1. [Stroustrup, 1997] Bjarne Stroustrup – The C++ Programming Language 3rd Edition, Addison Wesley, 1997 [Chapter 10]

2. [Stroustrup, 2013] Bjarne Stroustrup – The C++ Programming Language 4th Edition, Addison Wesley, 2013 [Chapter 16]