Lecture 7 - const, friend, this, static

Meng-Hsun Tsai
CSIE, NCKU

Introduction (cont.)

- const objects and const member functions prevent modifications of objects and enforce the principle of least privilege.
- Composition is a form of reuse in which a class can have objects of other classes as members.
- Friendship enables a class designer to specify nonmember functions that can access a class's non-public members
- The this pointer is an implicit argument to each of a class's non-static member functions. It allows those member functions to access the correct object's data members and other non-static member functions.
- static class members are class-wide members.

Copy Constructor

- Objects may be passed as function arguments and may be returned from functions.
- Such passing and returning is performed using pass-by-value by default—a copy of the object is passed or returned.
- C++ creates a new object and uses a copy constructor to copy the original object’s values into the new object.
- For each class, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.

Pass-by-*const*-reference

- Passing an object by value is good from a security stand point, because the called function has no access to the original object in the caller, but pass-by-value can degrade performance when making a copy of a large object.
- Pass-by-reference offers good performance but is weaker from a security standpoint, because the called function is given access to the original object.
- Pass-by-const-reference is a safe, good-performing alternative.

```
int func (const AClass & AnObject);
or
int func (AClass const & AnObject);
```

Copy Constructor vs. *Operator =*

```
#include <iostream>
using namespace std;
class Time {
public:
    Time(int h=0, int m=0):hour(h),minute(m) {}
    Time(const Time & t):
             hour(t.hour),minute(t.minute) {
        cout << "Copy Constructor" << hour
               << " " << minute << endl;
    }
    void operator = (Time const & t ) {
        cout << "Operator =" << hour
                << " " << minute << endl;
        hour = t.hour;
        minute = t.minute;
    }
private:
    int hour;
    int minute;
};

Time func(Time tt) {  return tt;  }
int main()
{
    Time t1(5, 10);
    Time t2(t1), t3;
    t3 = t2;
    t1 = func(t2);
    return 0;
}
```

```
Copy Constructor 5 10
Operator = 0 0
Copy Constructor 5 10
Copy Constructor 5 10
Operator = 5 10
```

const Objects and const Member Functions

- You may use keyword const to specify that an object is not modifiable and that any attempt to modify the object should result in a compilation error.
- Declaring variables and objects const when appropriate can improve performance. Compilers can perform certain optimizations on constants that cannot be performed on variables.
- Attempting to declare a constructor or destructor const is a compilation error.

const Objects and const Member Functions (cont.)

- C++ disallows member function calls for const objects unless the member functions themselves are also declared const (even for get member functions that do not modify the object).
- A member function is specified as const both in its prototype and in its definition.
- Defining as const a member function that modifies a data member of the object is a compilation error.
- Defining as const a member function that calls a non-const member function of the class on the same object is a compilation error.

Error: Trying to Modify const Objects

- Attempts to modify a `const` object are caught at compile time rather than causing execution-time errors.

```
#include <string>
using namespace std;
int main()
{
        const string Str1("NCKU is cool!");
        Str1 = "I love NCKU!";
        return 0;
}
```

```
> g++ -o mod_const_obj mod_const_obj.cpp
mod_const_obj.cpp: In function `int main()':
mod_const_obj.cpp:6: error: passing `const std::string' as `this' argument of `std::basic_string<_CharT, _Traits, _Alloc>& std::basic_string<_CharT, _Traits, _Alloc>::operator=(const _CharT*) [with _CharT = char, _Traits = std::char_traits<char>, _Alloc = std::allocator<char>]' discards qualifiers
```

Clock4.h and *Clock4.cpp*

Clock4.h

```
#ifndef CLOCK_H
#define CLOCK_H
#include <ctime>
using namespace std;
class Clock {
    public:
        Clock(clock_t s=0,
                    clock_t e=0);
        void start();
        void stop();
        void setStart(clock_t start_ts);
        clock_t getStart();
        double getElapsedTime()
             const;
    private:
        clock_t start_ts, elapsed_time;
};
#endif
```

Clock4.cpp

```
#include <iostream>
#include "Clock4.h"
using namespace std;
Clock::Clock(clock_t s, clock_t e):
      elapsed_time(e) { setStart(s); }
void Clock::start() { setStart(clock()); }
void Clock::stop() {
        elapsed_time = clock() - getStart(); }
void Clock::setStart(clock_t ts) {
       start_ts = (ts>0)?ts:clock(); }
clock_t Clock::getStart() {
    return start_ts;
}
double Clock::getElapsedTime() const {
      return (double)(elapsed_time) /
                    CLOCKS_PER_SEC ;
}
```

clocks6.cpp

```
#include <iostream>
#include "Clock4.h"
using namespace std;
int main()
{
    Clock clk;
    const Clock min_time(0, 10*128);

    cout << "min_time.start_ts =  "
        << min_time.getStart() << endl;
    cout << "wait until clk's elapsed time larger than "
        << min_time.getElapsedTime() << " seconds" << endl;
    clk.start();
    while(clk.getElapsedTime() <= min_time.getElapsedTime())
        clk.stop();
    cout << clk.getElapsedTime() << endl;

    return 0;
}
```

```
> g++ -o clocks6 clocks6.cpp
Clock4.cpp
clocks6.cpp: In function 'int
main()':
clocks6.cpp:10: error: passing
'const Clock' as 'this' argument
of 'clock_t Clock::getStart()'
discards qualifiers
```

Modified *clocks6.cpp*

```
#include <iostream>
#include "Clock4.h"
using namespace std;
int main()
{
    Clock clk;
    const Clock min_time(0, 10*128);

    cout << "wait until clk's elapsed time larger than "
              << min_time.getElapsedTime()
               << " seconds" << endl;
    clk.start();
    while(clk.getElapsedTime() <= min_time.getElapsedTime())
        clk.stop();
    cout << clk.getElapsedTime() << endl;

    return 0;
}
```

PASSED PASSED PASSED

```
wait until clk's elapsed time larger than 10
seconds
10.0078
```

Error: Data Assignment / non-*const* Function Call in a *const* Member Function

```
1  class Cls {
2      void const_func() const
3      {
4          data = 3;
5          non_const_func();
6      }
7      void non_const_func() {return ;}
8      int data;
9  };
```

```
> g++ -o const_memfunc const_memfunc.cpp
const_memfunc.cpp: In member function 'void Cls::const_func() const':
const_memfunc.cpp:4: error: assignment of data-member 'Cls::data' in read-only structure
const_memfunc.cpp:5: error: passing 'const Cls' as 'this' argument of 'void Cls::non_const_func()' discards qualifiers
```

Overloaded const Member Function

- A const member function can be overloaded with a non-const version.
- The compiler chooses which overloaded member function to use based on the object on which the function is invoked.
- If the object is const, the compiler uses the const version. If the object is not const, the compiler uses the non-const version.

Overloaded const Member Function (cont.)

```
#include <iostream>
using namespace std;
class Cls {
public:
    Cls():x(3){ }
    void func() const { cout << "const member function\n"; }
    void func() { cout << "non-const member function\n"; }
    int x;
};
int main()
{
    const Cls constObj;
    Cls nonConstObj;

    constObj.func();
    nonConstObj.func();
    return 0;
}
```

```
const member function
non-const member function
```

Member Initializer

- All data members can be initialized using member initializer, but const data members and data members that are references **must be** initialized using member initializers.
- Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body.
  - Separated from the parameter list with a colon (:).
  - Each member initializer consists of the data member name followed by parentheses containing the member's initial value.

```
Clock::Clock(clock_t s, clock_t e):elapsed_time(e) { setStart(s); }
```

Error: Initializing *const* Data Member in the Body of Constructor

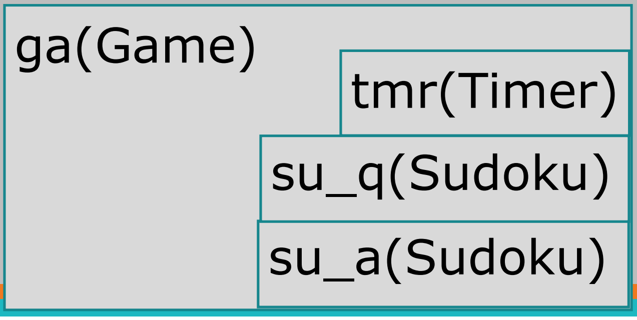
```
 1  #include <iostream>
 2  using namespace std;
 3  class Cls {
 4  public:
 5      Cls(){
 6          x = 3;
 7          y = 4;
 8      }
 9  private:
10      int x;
11      const int y;
12  };
13  int main()
14  {
15      const Cls obj;
16
17      return 0;
18  }
```

```
> g++ -o mem_init mem_init.cpp
mem_init.cpp: In constructor 'Cls::Cls()':
mem_init.cpp:5: error: uninitialized member
'Cls::y' with 'const' type 'const int'
mem_init.cpp:7: error: assignment of read-only
data-member 'Cls::y'
```

Composition: Objects as Members of Classes

- Composition
  - Sometimes referred to as a has-a relationship
  - A class can have objects of other classes as members
- An object's constructor can pass arguments to member-object constructors via member initializers.
- Member objects are constructed in the order in which they are declared in the class definition (not in the order they are listed in the constructor's member initializer list) and before their enclosing class objects (sometimes called host objects) are constructed.

composition.cpp



```cpp
#include <iostream>
using namespace std;
class Timer {
public:
    Timer(int a):x(a) { cout << "Timer ctor, x = " << x << endl; }
    ~Timer() { cout << "Timer dtor, x = " << x << endl; }
    int x;
};
class Sudoku {
public:
    Sudoku(int c):z(c) { cout << "Sudoku ctor, z = " << z << endl; }
    ~Sudoku() { cout << "Sudoku dtor, z = " << z << endl; }
    int z;
};
class Game {
public:
    Game(int p, const Sudoku & q, int r)
      :tmr(p),su_a(q),su_q(r)
      { cout << "Game ctor\n";    }
    ~Game() { cout << "Game dtor\n";    }
private:
    Timer tmr;
    Sudoku su_q, su_a;
};
int main()
{
    Sudoku su(5);
    Game ga(1,su,3);
    return 0;
}
```

```
Sudoku ctor, z = 5
Timer ctor, x = 1
Sudoku ctor, z = 3
Game ctor
Game dtor
Sudoku dtor, z = 5
Sudoku dtor, z = 3
Timer dtor, x = 1
Sudoku dtor, z = 5
```

Default Copy Constructor

- As you study class Sudoku, notice that the class does not provide a constructor that receives a parameter of type Sudoku.
- Why can the Game constructor's member initializer list initialize the su_a object by passing Sudoku objects to their Sudoku constructors?
- The compiler provides each class with a default copy constructor that copies each data member of the constructor's argument object into the corresponding member of the object being initialized.

Double Initialization

- If a member object is not initialized through a member initializer, the member object’s default constructor will be called implicitly.
- Initialize member objects explicitly through member initializers. This eliminates the overhead of “doubly initializing” member objects—once when the member object’s default constructor is called and again when set functions are called in the constructor body (or later) to initialize the member object.

friend Functions and friend Classes

- A friend function of a class is defined outside that class's scope, yet has the right to access the non-public (and public) members of the class.
- Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.
- Friendship is granted, not taken.
- The friendship relation is neither symmetric nor transitive.

friend Functions and friend Classes (cont.)

- Even though the prototypes for friend functions appear in the class definition, friends are not member functions.
- Member access notions of private, protected and public are not relevant to friend declarations, so friend declarations can be placed anywhere in a class definition.
- However, it is suggested to place all friendship declarations first inside the class definition's body and do not precede them with any access specifier.

Replacing Public Member Function Call by Direct Access in *friend* Function

sudoku_solve2.cpp

```
bool solve(Sudoku question,
           Sudoku & answer)
{
    …
    for(int num=1; num<=9; ++num)
    {
        question.map[firstZero]=num;
       // replace question.setElement();
        if(solve(question, answer))
            return true;
    }
int main()
{
 …
  for(int i=0;i<81;++i)  // read in question
  {
      infile >> num;
      question.map[i] = num;
       // replace question.setElement();
  }
```

Sudoku.h

```
class Sudoku {
    friend bool solve(Sudoku question,
                      Sudoku & answer);
    friend int main();
public:
```

infile (8 blanks)

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 0 3 4 5 6 7 8 9
1 2 3 4 5 0 7 8 9
1 2 3 4 5 6 7 0 9
1 2 0 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 0 7 8 9
0 2 3 4 0 6 7 0 9
```

```
> time ./sudoku_solve ;
Unsolvable!!
18.567u 0.007s 0:18.57
99.9%    10+2757k
0+0io 0pf+0w
> time ./sudoku_solve2
Unsolvable!!
17.830u 0.000s 0:17.83
100.0%   10+2757k
0+0io 0pf+0w
```

(18.567 - 17.83) / 18.567 = 3.97% (improved)

friend Class

```
#include <iostream>
using namespace std;
class A {
    friend class B;
private:
    int x;
};
class B{
public:
    void func(A & aa) {
        aa.x = 3; cout << aa.x;
    }
};
int main()
{
    A a;
    B b;
    b.func(a);
}
```

```
3
```

Error : *friend* Member Function of Unrecognized Class

```
 1  #include <iostream>
 2  using namespace std;
 3  class A {
 4      friend void B::func(A&);
 5  private:
 6      int x;
 7  };
 8  class B{
 9  public:
10      void func(A & aa) {
11          aa.x = 3;
12          cout << aa.x;
13      }
14  };
15  int main()
16  {
17      A a;
18      B b;
19      b.func(a);
20  }
```

```
>g++ -o friend_memfunc friend_memfunc.cpp
friend_memfunc.cpp:4: error: 'B' has not been declared
friend_memfunc.cpp: In member function 'void B::func(A&)':
friend_memfunc.cpp:6: error: 'int A::x' is private
friend_memfunc.cpp:11: error: within this context
friend_memfunc.cpp:6: error: 'int A::x' is private
friend_memfunc.cpp:12: error: within this context
```

friend Member Function of Another Class

```cpp
#include <iostream>
using namespace std;
class A;
class B{
public:
    void func(A & aa);
};
class A {
    friend void B::func(A &);
private:
    int x;
};
void B::func(A & aa) {
    aa.x = 3;
    cout << aa.x;
}
int main()
{
    A a;
    B b;
    b.func(a);
}
```

```
3
```

PASSED PASSED PASSED

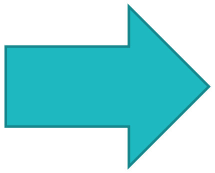
Type of the this Pointer

- How do member functions know *which* object's data members to manipulate? Every object has access to its own address through a pointer called this (a C++ keyword).
- The type of the this pointer depends on the type of the object and whether the member function in which this is used is declared const.

```
Timer::start ( )
{start_ts=time(0); }

Timer tmr;
tmr.start();
```

Compiler



```
Timer::start(Timer * const this)
{this -> start_ts = time(0);}

Timer tmr;
Timer::start( & tmr);
```

Clock5.h and Clock5.cpp

Clock5.h

```
#ifndef CLOCK_H
#define CLOCK_H
#include <ctime>
using namespace std;
class Clock {
    public:
        Clock(clock_t s=0, clock_t e=0);
        Clock & start();
        Clock & stop();
        void setStart(clock_t start_ts);
        clock_t getStart();
        double getElapsedTime() const;
    private:
        clock_t start_ts, elapsed_time;
};
#endif
```

Clock5.cpp

```
#include <iostream>
#include "Clock5.h"
using namespace std;
Clock::Clock(clock_t s, clock_t e):
        elapsed_time(e) { setStart(s); }
Clock & Clock::start() {
    this->setStart(clock());
    return (*this);
}
Clock & Clock::stop() {
    (*this).elapsed_time = clock() - getStart();
    return (*this);
}
void Clock::setStart(clock_t ts) {
    start_ts = (ts>0)?ts:clock(); }
clock_t Clock::getStart() { return start_ts; }
double Clock::getElapsedTime() const {
    return (double)(elapsed_time) /
                    CLOCKS_PER_SEC ;
}
```

clocks7.cpp

```
#include <iostream>
#include "Clock5.h"
using namespace std;
int main()
{
    Clock clk;

    cout << clk.start().getElapsedTime() << endl;
    for(int i=0;i<10000000;++i)
        ;
    cout << clk.stop().getElapsedTime() << endl;

    return 0;
}
```

```
0
0.257812
```

Using the this Pointer

- The this pointer is not part of the object itself.
- The this pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions.
- Objects use the this pointer implicitly or explicitly to reference their data members and member functions.

```
start_ts=time(0);
this->start_ts = time(0);
(*this).start_ts = time(0);
```

implicitly

explicitly

Cascaded Member Function Call

- Another use of the this pointer is to enable cascaded member-function calls (invoking multiple functions in the same statement).
- Why does the technique of returning *this as a reference work? The dot operator (.) associates from left to right, so line 8 first evaluates clk.start(), then returns a reference to object clk as the value of this function call.
- The remaining expression is then interpreted as clk.getElapsedTime( ).

```
clk.start().getElapsedTime();
```

static Class Members

- In certain cases, only one copy of a variable should be shared by all objects of a class.
- A static data member is used for these and other reasons (e.g., save storage).
- Such a variable represents “class-wide” information.
- Although they may seem like global variables, a class’s static data members have class scope.
- A fundamental-type static data member is initialized by default to 0.

static Class Members (cont.)

- A static const data member can be initialized in its declaration in the class definition.
- All other static data members must be defined at global namespace scope and can be initialized only in those definitions.

```
class Cls {
public:       Cls(){  NumObject++;  }
              static int NumObject;
              static const int MaxNum = 100;
};
int Cls::NumObject = 0;
int main()
{
    ...
```

static Class Members (cont.)

- A class's static members exist and can be used even when no objects of that class exist.
- To access a public static class member when no objects of the class exist, prefix the class name and the binary scope resolution operator (::) to the name of the data member.
- To access a private or protected static class member when no objects of the class exist, provide a public static member function and call the function by prefixing its name with the class name and binary scope resolution operator.

static Class Members (cont.)

- It is a compilation error to include keyword static in the definition of a static data member at global namespace scope.

```
class Cls {
public:      Cls(){  NumObject++;  }
             static int NumObject;
};
int Cls::NumObject = 0;
int main()
{
...
```

Just Declaration

Definition (Do not use “static” here.)

Clock6.h and *Clock6.cpp*

Clock6.h

```
#ifndef CLOCK_H
#define CLOCK_H
#include <ctime>
using namespace std;
class Clock {
   public:
      Clock();
      Clock(const Clock&);
      ~Clock();
      void start();
      void stop();
      double getElapsedTime() const;
      static int getNum();
      static clock_t getTotal();
   private:
      clock_t start_ts, elapsed_time;
      static int numClock;
      static clock_t totalClock;
};
#endif
```

Clock6.cpp

```
#include <iostream>
#include "Clock6.h"
using namespace std;
int Clock::numClock = 0;
clock_t Clock::totalClock = 0;
Clock::Clock() {++numClock; }
Clock::Clock(const Clock&) {++numClock; }
Clock::~Clock() {--numClock; }
void Clock::start() { start_ts=clock(); }
void Clock::stop() {
   elapsed_time = clock() - start_ts;
   totalClock += elapsed_time;
}
double Clock::getElapsedTime() const {
   return (double)(elapsed_time) /
               CLOCKS_PER_SEC ;
}
int Clock::getNum() { return numClock; }
clock_t Clock::getTotal() { return totalClock; }
```

clocks8.cpp

```
#include <vector>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include "Clock6.h"
using namespace std;
int main()
{
   srandom(time(NULL));
   vector<Clock> v_clk(5);
   long int counter;
   for(int i=0;i<5;++i)
   {
       v_clk[i].start();
       counter = 10000000+random();
       for(long int j=0;j<counter;++j)
           ;
       v_clk[i].stop();
   }
   for(int i=0;i<5;++i)
   {
       cout << v_clk[i].getElapsedTime()
                << endl;
   }
   cout<<“There are “<<v_clk[0].getNum()
           << " clocks.\n";
   cout << “Average Time: ” << (double)
           Clock::getTotal() / Clock::getNum() /
           CLOCKS_PER_SEC << endl;

   return 0;
}
```

```
0.75
3.42188
2.39844
3.9375
4.11719
There are 5 clocks.
Average Time: 2.925
```

this Pointer vs. static Member Function

- A member function should be declared static if it does not access non-static data members or non-static member functions of the class.
- A static member function does not have a this pointer, because static data members and static member functions exist independently of any objects of a class.
- The this pointer must refer to a specific object of the class, and when a static member function is called, there might not be any objects of its class in memory.

this Pointer vs. static Member Function (cont.)

- Using the this pointer in a static member function is a compilation error.
- Declaring a static member function const is a compilation error.

```
#include <iostream>
using namespace std;
class Cls {
   static const int x = 5;
   static void func() { cout << this; }
   static void func2() const {}
};
int main()  {     return 0;  }
```

```
> g++ -o static_const_memfunc static_const_memfunc.cpp
static_const_memfunc.cpp:6: error: static member function 'static void
Cls::func2()' cannot have cv-qualifier
static_const_memfunc.cpp: In static member function 'static void Cls::func()':
static_const_memfunc.cpp:5: error: 'this' is unavailable for static member functions
```