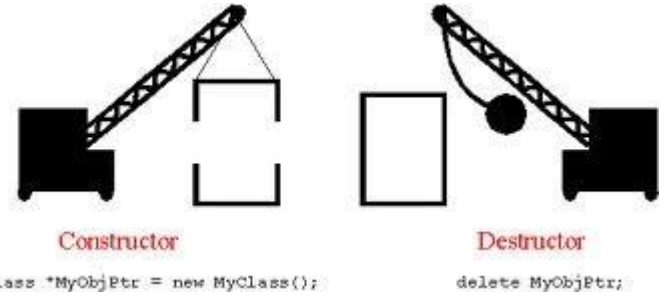
Lecture 6 - Scope, Constructor and Descructor

Meng-Hsun Tsai
CSIE, NCKU



Introduction

- Class scope and the relationships among class members.
- Accessing a class's public members via three types of "handles"—the name of an object, a reference to an object or a pointer to an object.
- How default arguments can be used in a constructor.

Introduction (cont.)

- Destructors that perform “termination housekeeping” on objects before they are destroyed.
- The order in which constructors and destructors are called.
- The dangers of member functions that return references to private data.
- Default memberwise assignment for copying the data members in the object on the right side of an assignment into the corresponding data members of the object on the left side of the assignment.

Class Scope

- Even though a member function declared in a class definition may be defined outside that class definition, that member function is still within that class's scope.
- A class's data members and member functions belong to that class's scope.
- Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name.

Clock.h

```
 3  class Clock {
    …
11      private:
12          clock_t start_ts;
13  };
```

Clock::start_ts

Clock.cpp

```
 2  Clock::Clock() { setStart(0); }
    …
 9  void Clock::setStart(clock_t ts) {
10      start_ts = (ts>0)?ts:clock();
11  }
```

Clock::Clock()

Clock::setStart()

Scopes

```
namespace ns {
    int y;
}
Class Human {
    Human();
    int class_var;
}
Human::Human() {
    class_var = 3;
}
int main ()
{
label:
   int func_var;
   {
       int block_var;
   }
}
```

Namespace scope (ns::y)

Class scope (Human::class_var, Human::Human())

Function scope (func_var, label)

Block (Local) scope (block_var)

Namespace Scope

```
#include <iostream>

namespace std {
int main()
{
    cout << "kerker" << endl ;
}
}

int main()
{
    std::main();
}
```

Namespace scope (std::main())

Global namespace scope (main())

Accessing Class Members Through *Name*, *Pointer* and *Reference*

- Outside a class's scope, public class members are referenced through one of the handles on an object—an object name, a reference to an object or a pointer to an object.
- The dot member selection operator (.) is preceded by an object's name or with a reference to an object to access the object's members.
- The arrow member selection operator (->) is preceded by a pointer to an object to access the object's members.

Accessing Class Members Through *Name*, *Pointer* and *Reference* (cont.)

Clock.h

```
 1  #ifndef CLOCK_H
 2  #define CLOCK_H
 3  #include <ctime>
 4  using namespace std;
 5  class Clock {
 6      public:
 7          Clock();
 8          Clock(clock_t s);
 9          void start();
10          void stop();
11          void setStart(clock_t start_ts);
12          clock_t getStart();
13          double getElapsedTime();
14      private:
15          clock_t start_ts, elapsed_time;
16  };
17  #endif
```

Clock.cpp

```
 1  #include "Clock.h"
 2  Clock::Clock() { setStart(0); }
 3  Clock::Clock(clock_t s) { setStart(s); }
 4  void Clock::start() {
 5      setStart(clock());
 6  }
 7  void Clock::stop() {
 8      elapsed_time = clock() - getStart();
 9  }
10  void Clock::setStart(clock_t ts) {
11      start_ts = (ts>0)?ts:clock();
12  }
13  clock_t Clock::getStart() {
14      return start_ts;
15  }
16  double Clock::getElapsedTime() {
17      return (double)(elapsed_time) /
                                CLOCKS_PER_SEC ;
18  }
```

Accessing Class Members Through *Name*, *Pointer* and *Reference* (cont.)

```
 1  #include <iostream>
 2  #include "Clock.h"
 3  using namespace std;
 4
 5  int main()
 6  {
 7      Clock clk;
 8      Clock* clk_ptr = &clk;
 9      Clock& clk_ref = clk;
10
11      clk.start();
12      for(int j=0;j<100000000;++j)
13          ;
14      clk.stop();
15      cout << "clk.elapsed_time = "
        <<clk.getElapsedTime()<<endl;
16
17      clk_ptr->start();
18      for(int j=0;j<100000000;++j)
19          ;
20      clk_ptr->stop();
21      cout << "clk_ptr->elapsed_time = " <<
                clk_ptr->getElapsedTime() << endl;
22
23      clk_ref.start();
24      for(int j=0;j<100000000;++j)
25          ;
26      clk_ref.stop();
27      cout << "clk_ref.elapsed_time = " <<
                clk_ref.getElapsedTime() << endl;
28
29      return 0;
30  }
```

```
clk.elapsed_time = 0.164062
clk_ptr->elapsed_time = 0.15625
clk_ref.elapsed_time = 0.164062
```

Constructors with Default Arguments

- Like other functions, constructors can specify default arguments.
- A constructor that defaults all its arguments is also a default constructor—i.e., a constructor that can be invoked with no arguments.
- There can be at most one default constructor per class.
- Any change to the default argument values of a function requires the client code to be recompiled (to ensure that the program still functions correctly).

Clock.h and *Clock.cpp*

Clock.h

```
 1  #ifndef CLOCK_H
 2  #define CLOCK_H
 3  #include <ctime>
 4  using namespace std;
 5  class Clock {
 6      public:
 7          Clock(clock_t s=0,
                  clock_t e=0);
 8          void start();
 9          void stop();
10          void setStart(clock_t start_ts);
11          clock_t getStart();
12          double getElapsedTime();
13      private:
14          clock_t start_ts, elapsed_time;
15  };
16  #endif
```

Clock.cpp

```
 1  #include "Clock.h"
 2  Clock::Clock(clock_t s, clock_t e) {
 3      setStart(s);
 4      elapsed_time = e;
 5  }
 6  void Clock::start() { setStart(clock()); }
 7  void Clock::stop() {
 8      elapsed_time = clock() - getStart();
 9  }
10  void Clock::setStart(clock_t ts) {
11      start_ts = (ts>0)?ts:clock();
12  }
13  clock_t Clock::getStart() {
14      return start_ts;
15  }
16  double Clock::getElapsedTime() {
17      return (double)(elapsed_time) /
                          CLOCKS_PER_SEC ;
18  }
```

clocks2.cpp

```
 1 #include <iostream>
 2 #include "Clock.h"
 3 using namespace std;
 4
 5 int main()
 6 {
 7     Clock clk;
 8     Clock clk2(5);
 9     Clock clk3(3,5);
10     cout << "clk.start_ts = " << clk.getStart() << endl;
11     cout << "clk.elapsed_time = " << clk.getElapsedTime() << endl;
12     cout << "clk2.start_ts = " << clk2.getStart() << endl;
13     cout << "clk2.elapsed_time = " << clk2.getElapsedTime() << endl;
14     cout << "clk3.start_ts = " << clk3.getStart() << endl;
15     cout << "clk3.elapsed_time = " << clk3.getElapsedTime() << endl;
16
17     return 0;
18 }
```

```
clk.start_ts = 0
clk.elapsed_time = 0
clk2.start_ts = 5
clk2.elapsed_time = 0
clk3.start_ts = 3
clk3.elapsed_time = 0.0390625
```

Destructors

- The name of the destructor for a class is the tilde character (~) followed by the class name.

```
8          ~Clock();
```

- Often referred to with the abbreviation "dtor" in the literature.
- Called implicitly when an object is destroyed.
- The destructor itself does not actually release the object's memory—it performs termination housekeeping before the object's memory is reclaimed, so the memory may be reused to hold new objects.
- Receives no parameters and returns no value.
- Can not specify a return type—not even void.

Destructors (cont.)

- A class may have only one destructor.
- A destructor must be public.
- If you do not explicitly provide a destructor, the compiler creates an "empty" destructor.
- It's a syntax error to attempt to pass arguments to a destructor, to specify a return type for a destructor (even void cannot be specified), to return values from a destructor or to overload a destructor.

Clock2.h and *Clock2.cpp*

Clock2.h

```
 1  #ifndef CLOCK_H
 2  #define CLOCK_H
 3  #include <ctime>
 4  using namespace std;
 5  class Clock {
 6      public:
 7          Clock(clock_t s=0,
                    clock_t e=0);
 8          ~Clock();
 9          void start();
10          void stop();
11          void setStart(clock_t start_ts);
12          clock_t getStart();
13          double getElapsedTime();
14      private:
15          clock_t start_ts, elapsed_time;
16  };
17  #endif
```

Clock2.cpp

```
 1  #include <iostream>
 2  #include "Clock2.h"
 3  using namespace std;
 4  Clock::Clock(clock_t s, clock_t e) {
 5      setStart(s); elapsed_time = e;
 6      cout << "ctor, start = " << s << endl;
 7  }
 8  Clock::~Clock() {
 9      cout << "dtor, start = " <<
              this->getStart() << endl;
10  }
11  void Clock::start() { setStart(clock()); }
12  void Clock::stop() {
        elapsed_time = clock() - getStart(); }
13  void Clock::setStart(clock_t ts) {
        start_ts = (ts>0)?ts:clock(); }
14  clock_t Clock::getStart() { return start_ts; }
15  double Clock::getElapsedTime() { return
    (double)(elapsed_time) / CLOCKS_PER_SEC ; }
```

clocks3.cpp

```
 1  #include <iostream>
 2  #include "Clock2.h"
 3  using namespace std;
 4  void func();
 5  Clock first(1);
 6  int main()
 7  {
 8      Clock second(2);
 9      static Clock third(3);
10      func();
11      Clock fourth(4);
12      return 0;
13  }
14  void func()
15  {
16      Clock fifth(5);
17      static Clock sixth(6);
18      Clock seventh(7);
19  }
```

```
ctor, start = 1
ctor, start = 2
ctor, start = 3
ctor, start = 5
ctor, start = 6
ctor, start = 7
dtor, start = 7
dtor, start = 5
ctor, start = 4
dtor, start = 4
dtor, start = 2
dtor, start = 6
dtor, start = 3
dtor, start = 1
```

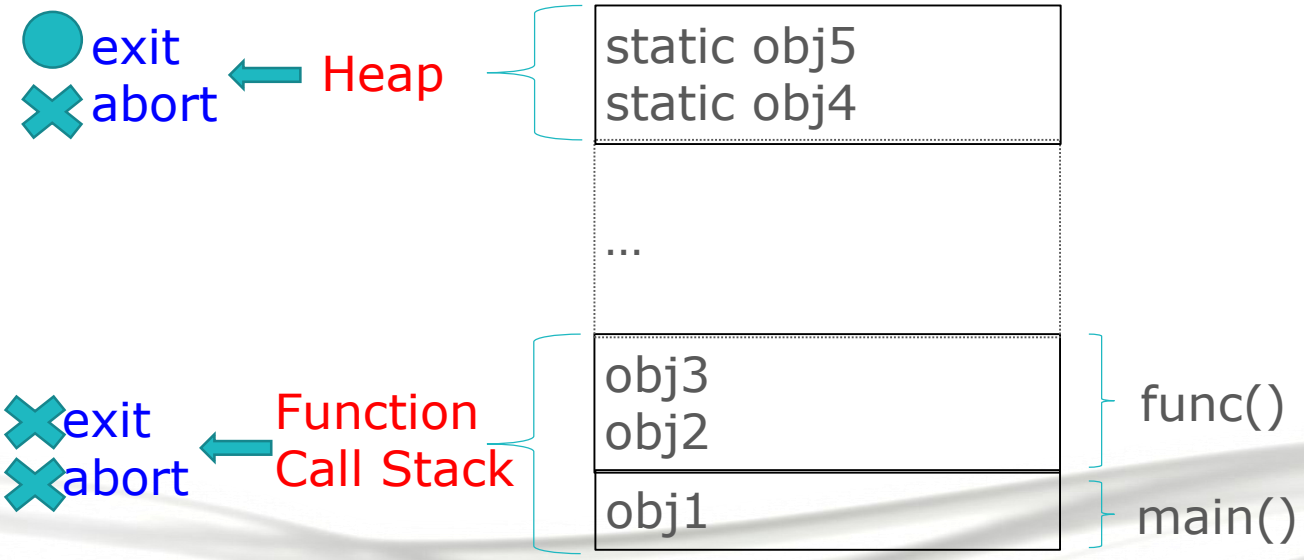
```
first
main()
  second
  static third
  func()
    fifth
    static sixth
    seventh
  fourth
```

When Constructors and Destructors Are Called

- Constructors and destructors are called implicitly.
- The order in which these function calls occur depends on the order in which execution enters and leaves the scopes where the objects are instantiated.
- Generally, destructor calls are made in the reverse order of the corresponding constructor calls
- Constructors are called for objects defined in global scope before any other function (including main) in that file begins execution (although the order of execution of global object constructors between files is not guaranteed).
- The corresponding destructors are called when main terminates.

When Constructors and Destructors Are Called (cont.)

- Function exit forces a program to terminate immediately and does not execute the destructors of automatic objects.
- Function abort performs similarly to function exit but forces the program to terminate immediately, without allowing the destructors of any objects to be called.



When Constructors and Destructors Are Called (cont.)

- The constructor for a static local object is called only once, when execution first reaches the point where the object is defined—the corresponding destructor is called when main terminates or the program calls function exit.
- Global and static objects are destroyed in the reverse order of their creation.

A Subtle Trap—Returning a Reference to a `private` Data Member

- A reference to an object is an alias for the name of the object and, hence, may be used on the left side of an assignment statement (*lvalue*).
- Unfortunately a public member function of a class can return a reference to a private data member of that class. In this case, the returned private data member can be directly modified outside.

Clock3.h and *Clock3.cpp*

Clock3.h

```
 1  #ifndef CLOCK_H
 2  #define CLOCK_H
 3  #include <ctime>
 4  using namespace std;
 5  class Clock {
 6      public:
 7          Clock(clock_t s=0,
                    clock_t e=0);
 8          void start();
 9          void stop();
10          void setStart(clock_t start_ts);
11          clock_t getStart();
12          double getElapsedTime();
13          clock_t & badFunc();
14      private:
15          clock_t start_ts, elapsed_time;
16  };
17  #endif
```

Clock3.cpp

```
 1  #include <iostream>
 2  #include "Clock3.h"
 3  using namespace std;
 4  Clock::Clock(clock_t s, clock_t e) {
 5      setStart(s); elapsed_time = e;
 6  }
 7  clock_t & Clock::badFunc() { return start_ts;}
 8  void Clock::start() { setStart(clock()); }
 9  void Clock::stop() {
        elapsed_time = clock() - getStart(); }
10  void Clock::setStart(clock_t ts) {
        start_ts = (ts>0)?ts:0; }
11  clock_t Clock::getStart() { return start_ts; }
12  double Clock::getElapsedTime() { return
    (double)(elapsed_time) / CLOCKS_PER_SEC ; }
```

clocks4.cpp

```
 1  #include <iostream>
 2  #include "Clock3.h"
 3  using namespace std;
 4  int main()
 5  {
 6      Clock clk(-5);
 7
 8      cout << "clk.start = " << clk.getStart() << endl;
 9
10      clock_t &ts_ref = clk.badFunc();
11      ts_ref = -8;
12      cout << "clk.start = " << clk.getStart() << endl;
13
14      clk.badFunc() = -3;
15      cout << "clk.start = " << clk.getStart() << endl;
16
17      return 0;
18  }
```

```
clk.start = 0
clk.start = -8
clk.start = -3
```

Default Memberwise Assignment

- The assignment operator (=) can be used to assign an object to another object of the same type.
- By default, such assignment is performed by memberwise assignment: Each data member of the object on the right of the assignment operator is assigned individually to the same data member in the object on the left of the assignment operator.
- *Caution:* Memberwise assignment can cause serious problems when used with a class whose data members contain pointers to dynamically allocated memory.

clocks5.cpp

```
 1  #include <iostream>
 2  #include "Clock3.h"
 3  using namespace std;
 4  int main()
 5  {
 6      Clock clk(3,5);
 7      Clock clk2;
 8      cout << "clk.start = " << clk.getStart()
 9           << ", clk.getElapsedTime = " << clk.getElapsedTime() << endl;
10
11      cout << "clk2.start = " << clk2.getStart()
12           << ", clk2.getElapsedTime = " << clk2.getElapsedTime() << endl;
13      clk2 = clk;
14      cout << "clk2.start = " << clk2.getStart()
15           << ", clk2.getElapsedTime = " << clk2.getElapsedTime() << endl;
16
17      return 0;
18  }
```

```
clk.start = 3, clk.getElapsedTime = 0.0390625
clk2.start = 0, clk2.getElapsedTime = 0
clk2.start = 3, clk2.getElapsedTime = 0.0390625
```