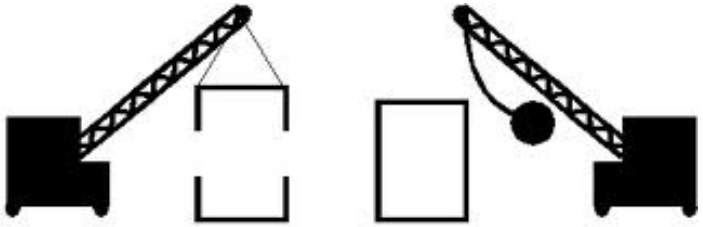
Class: A Deeper Look

Meng-Hsun Tsai
CSIE, NCKU



Introduction

- "Preprocessor wrappers" in header files to prevent the code in the header from being included into the same source code file more than once.
- 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.

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.

Preprocessor Wrapper

Sudoku.h

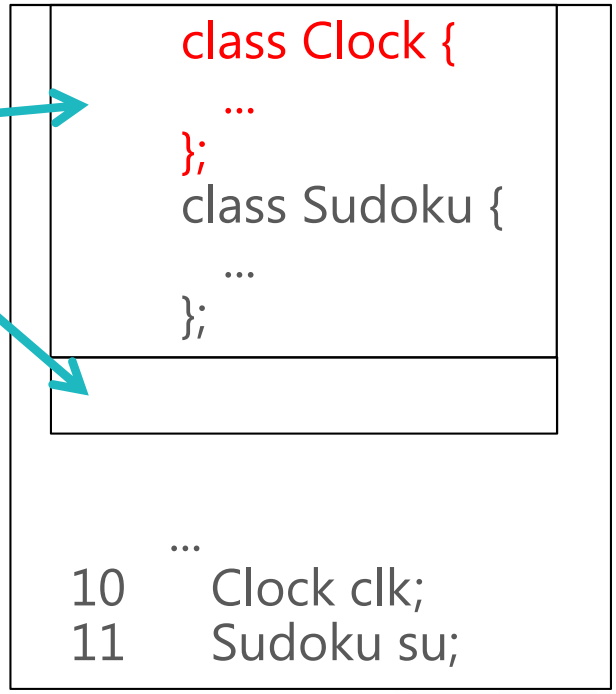
```
1  #ifndef SUDOKU_H
2  #define SUDOKU_H
3  #include "Clock.h"
4  class Sudoku {
     ...
20 };
21 #endif
```

Clock.h

```
1  #ifndef CLOCK_H
2  #define CLOCK_H
3  class Clock {
     ...
15 };
16 #endif
```

main.cpp

```
1  #include "Sudoku.h"
2  #include "Clock.h "
     ...
10     Clock clk;
11     Sudoku su;
     ...
```



Preprocessor Wrapper (cont.)

- The clock class definition is enclosed in the following preprocessor wrapper:

```
#ifndef CLOCK_H
#define CLOCK_H
   ...
#endif
```

- This prevents the code between `#ifndef` and `#endif` from being included if the name `CLOCK_H` has been defined.
- If the header has not been included previously in a file, the name `CLOCK_H` is defined by the `#define` directive and the header file statements are included.
- If the header has been included previously, `CLOCK_H` is defined already and the header file is not included again.

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 seven(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
```

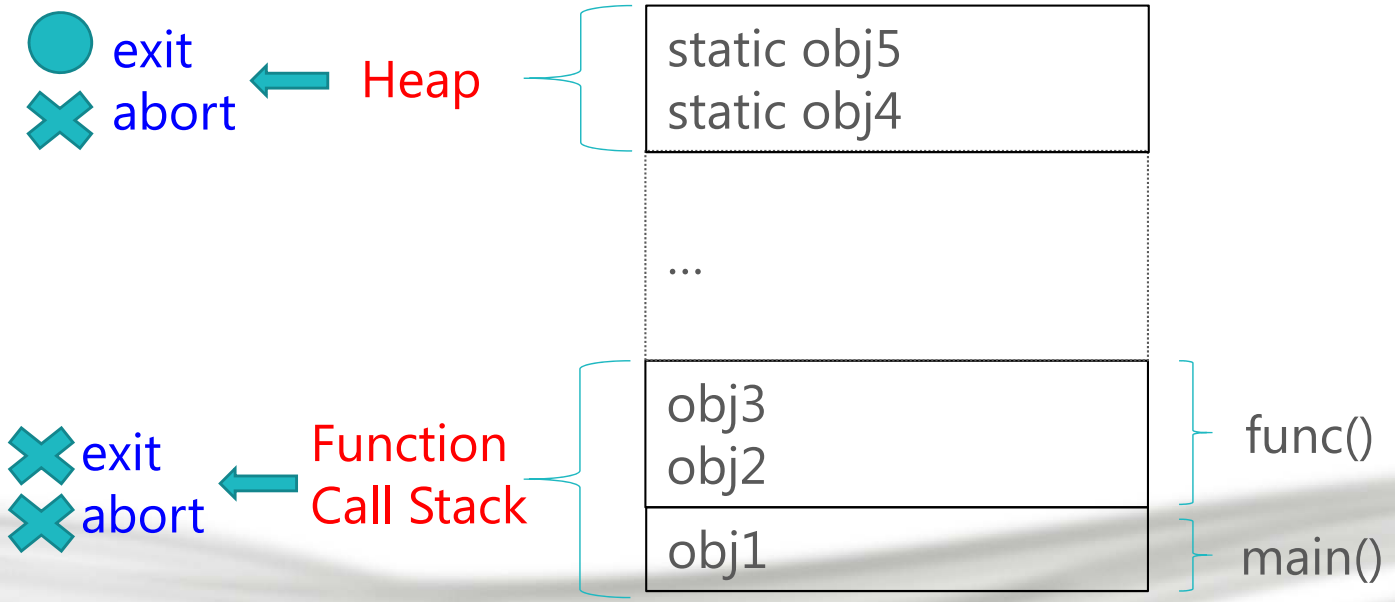
```
first
main()
  second
  static third
  func()
    fifth
    static sixth
    seventh
  fourth
```

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:clock(); }
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
```

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 =*

```
 1  #include <iostream>
 2  using namespace std;
 3  class Time {
 4  public:
 5     Time(int h=0, int m=0):hour(h),minute(m) {}
 6     Time(const Time & t):
              hour(t.hour),minute(t.minute) {
 7        cout << "Copy Constructor" << endl;
 8     }
 9     Time & operator = (Time const & t ) {
10        cout << "Operator =" << endl;
11        hour = t.hour;
12        minute = t.minute;
13     }
14  private:
15     int hour;
16     int minute;
17  };
18
```

```
19  Time func(Time tt) {  return tt;  }
20  int main()
21  {
22     Time t1(5, 10);
23     Time t2(t1), t3;
24     t3 = t2;
25     t1 = func(t2);
26     return 0;
27  }
```

```
Copy Constructor
Operator =
Copy Constructor
Copy Constructor
Operator =
```

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.


```
1 #include <string>
2 using namespace std;
3 int main()
4 {
5       const string Str1("NCKU is cool!");
6       Str1 = "I love NCKU!";
7       return 0;
8 }
```

```
> g++ -o mod_const_obj mod_const_obj.cpp
mod_const_obj.cpp: In function `int main()':
mod_const_obj.cpp:8: 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

```
 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()
             const;
13     private:
14        clock_t start_ts, elapsed_time;
15  };
16  #endif
```

Clock4.cpp

```
 1  #include <iostream>
 2  #include "Clock4.h"
 3  using namespace std;
 4  Clock::Clock(clock_t s, clock_t e):
       elapsed_time(e) { setStart(s); }
 5  void Clock::start() { setStart(clock()); }
 6  void Clock::stop() {
        elapsed_time = clock() - getStart(); }
 7  void Clock::setStart(clock_t ts) {
       start_ts = (ts>0)?ts:clock(); }
 8  clock_t Clock::getStart() {
 9     return start_ts;
10  }
11  double Clock::getElapsedTime() const {
12        return (double)(elapsed_time) /
                    CLOCKS_PER_SEC ;
13  }
```

clocks6.cpp

```
 1  #include <iostream>
 2  #include "Clock4.h"
 3  using namespace std;
 4  int main()
 5  {
 6     Clock clk;
 7     const Clock min_time(0, 10*128);
 8
 9     cout << "min_time.start_ts =  "
10          << min_time.getStart() << endl;
11     cout << "wait until clk's elapsed time larger than "
12          << min_time.getElapsedTime() << " seconds" << endl;
13     clk.start();
14     while(clk.getElapsedTime() <= min_time.getElapsedTime())
15        clk.stop();
16     cout << clk.getElapsedTime() << endl;
17
18     return 0;
19  }
```

```
> 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*

```
 1 #include <iostream>
 2 #include "Clock4.h"
 3 using namespace std;
 4 int main()
 5 {
 6    Clock clk;
 7    const Clock min_time(0, 10*128);
 8
 9    cout << "wait until clk's elapsed time larger than "
10         << min_time.getElapsedTime()
           << " seconds" << endl;
11    clk.start();
12    while(clk.getElapsedTime() <= min_time.getElapsedTime())
13       clk.stop();
14    cout << clk.getElapsedTime() << endl;
15
16    return 0;
17 }
```


```
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.)

```
 1  #include <iostream>
 2  using namespace std;
 3  class Cls {
 4  public:
 5      Cls():x(3){ }
 6      void func() const { cout << "const member function\n"; }
 7      void func() { cout << "non-const member function\n"; }
 8      int x;
 9  };
10  int main()
11  {
12      const Cls constObj;
13      Cls nonConstObj;
14
15      constObj.func();
16      nonConstObj.func();
17      return 0;
18  }
```

```
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.

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

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

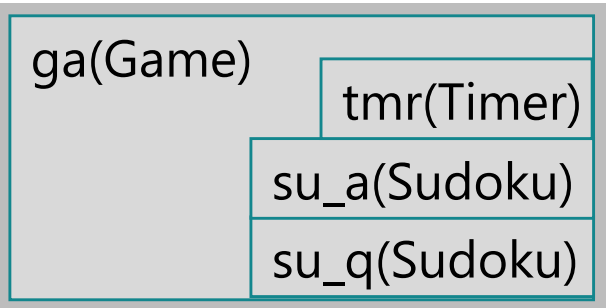
```
 1 #include <iostream>
 2 using namespace std;
 3 class Cls {
 4 public:
 5     Cls(){
 6         x = 3;
 7         y = 4;
 8     }
 9 private:
10     int x;
11     const int y;
12 };
13 int main()
14 {
15     const Cls obj;
16
17     return 0;
18 }
```


```
> 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




```
 1 #include <iostream>
 2 using namespace std;
 3 class Timer {
 4 public:
 5     Timer(int a):x(a) { cout <<
       "Timer ctor, x = " << x << endl; }
 6     ~Timer() { cout <<
       "Timer dtor, x = " << x << endl; }
 7     int x;
 8 };
 9 class Sudoku {
10 public:
11     Sudoku(int c):z(c) { cout <<
     "Sudoku ctor, z = " << z << endl; }
12     ~Sudoku() { cout <<
     "Sudoku dtor, z = " << z << endl; }
13     int z;
14 };
```

```
15 class Game {
16 public:
17     Game(int p, const Sudoku & q, int r)
        :tmr(p),su_a(q),su_q(r)
        { cout << "Game ctor\n";    }
18     ~Game() { cout << "Game dtor\n";    }
19 private:
20     Timer tmr;
21     Sudoku su_q, su_a;
22 };
23 int main()
24 {
25     Sudoku su(5);
26     Game ga(1,su,3);
27     return 0;
28 }
```

```
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

```
 7  bool solve(Sudoku question,
              Sudoku & answer)
 8  {
      ...
23      for(int num=1; num<=9; ++num)
24      {
25        question.map[firstZero]=num;
          // replace question.setElement();
26        if(solve(question, answer))
27          return true;
28      }
32  int main()
33  {
  ...
39    for(int i=0;i<81;++i)  // read in question
40    {
41      infile >> num;
42      question.map[i] = num;
        // replace question.setElement();
43  }
```

Sudoku.h

```
3 class Sudoku {
4    friend bool solve(Sudoku question,
                       Sudoku & answer);
5    friend int main();
6 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

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

```
3
```

Error : *friend* Member Function of Unrecognized Class

```
 1  #include <iostream>
 2  using namespace std;
 3  class A {
 4      friend 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:4: error: ISO C++ forbids declaration of
'func' with no type
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

```
 1  #include <iostream>
 2  using namespace std;
 3  class A;
 4  class B {
 5  public:
 6      void func(A & aa);
 7  };
 8  class A {
 9      friend void B::func(A &);
10  private:
11      int x;
12 };
13 void B::func(A & aa) {
14     aa.x = 3;
15     cout << aa.x;
16 }
17 int main()
18 {
19     A a;
20     B b;
21     b.func(a);
22 }
```

```
3
```

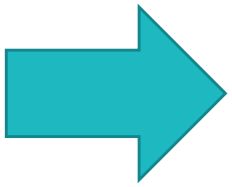
Type of the *this* Pointer

- 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

```
 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 & start();
 9        Clock & stop();
10        void setStart(clock_t start_ts);
11        clock_t getStart();
12        double getElapsedTime() const;
13     private:
14        clock_t start_ts, elapsed_time;
15  };
16  #endif
```

Clock5.cpp

```
 1  #include <iostream>
 2  #include "Clock5.h"
 3  using namespace std;
 4  Clock::Clock(clock_t s, clock_t e):
          elapsed_time(e) { setStart(s); }
 5  Clock & Clock::start() {
 6     this->setStart(clock());
 7     return (*this);
 8  }
 9  Clock & Clock::stop() {
10     (*this).elapsed_time = clock() - getStart();
11     return (*this);
12  }
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() const {
16     return (double)(elapsed_time) /
                  CLOCKS_PER_SEC ;
17  }
```

clocks7.cpp

```
 1  #include <iostream>
 2  #include "Clock5.h"
 3  using namespace std;
 4  int main()
 5  {
 6      Clock clk;
 7
 8      cout << clk.start().getElapsedTime() << endl;
 9      for(int i=0;i<10000000;++i)
10          ;
11      cout << clk.stop().getElapsedTime() << endl;
12
13      return 0;
14  }
```

```
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);
```

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 object of that class exists.
- To access a `public static` class member when no object of the class exists, 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

```
 1  #ifndef CLOCK_H
 2  #define CLOCK_H
 3  #include <ctime>
 4  using namespace std;
 5  class Clock {
 6     public:
 7        Clock();
 8        Clock(const Clock&);
 9        ~Clock();
10        void start();
11        void stop();
12        double getElapsedTime() const;
13        static int getNum();
14        static clock_t getTotal();
15     private:
16        clock_t start_ts, elapsed_time;
17        static int numClock;
18        static clock_t totalClock;
19  };
20  #endif
```

Clock6.cpp

```
 1  #include <iostream>
 2  #include "Clock6.h"
 3  using namespace std;
 4  int Clock::numClock = 0;
 5  clock_t Clock::totalClock = 0;
 6  Clock::Clock() {++numClock; }
 7  Clock::Clock(const Clock&) {++numClock; }
 8  Clock::~Clock() {--numClock; }
 9  void Clock::start() { start_ts=clock(); }
10  void Clock::stop() {
11     elapsed_time = clock() - start_ts;
12     totalClock += elapsed_time;
13  }
14  double Clock::getElapsedTime() const {
15     return (double)(elapsed_time) /
                  CLOCKS_PER_SEC ;
16  }
17  int Clock::getNum() { return numClock; }
18  clock_t Clock::getTotal() { return totalClock; }
```

clocks8.cpp

```
 1 #include <vector>
 2 #include <iostream>
 3 #include <ctime>
 4 #include <cstdlib>
 5 #include "Clock6.h"
 6 using namespace std;
 7 int main()
 8 {
 9    srandom(time(NULL));
10    vector<Clock> v_clk(5);
11    long int counter;
12    for(int i=0;i<5;++i)
13    {
14       v_clk[i].start();
15       counter = 10000000+random();
16       for(long int j=0;j<counter;++j)
17          ;
18       v_clk[i].stop();
19    }
20    for(int i=0;i<5;++i)
21    {
22       cout << v_clk[i].getElapsedTime()
                 << endl;
23    }
24    cout<<  "There are   "<<v_clk[0].getNum()
            << " clocks.\n";
25    cout <<   "Average Time: "   << (double)
          Clock::getTotal() / Clock::getNum() /
          CLOCKS_PER_SEC << endl;
26
27    return 0;
28 }
```

```
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 object 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.

```
1  #include <iostream>
2  using namespace std;
3  class Cls {
4     static const int x = 5;
5     static void func() { cout << this; }
6     static void func2() const {}
7  };
8  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
```