Object-Oriented Programming
(in C++)

Classes and Objects (Part II)

Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

Different Kinds of Members

- C++ has many choices of members
  - `const` members: members that require initialization and cannot be updated
    - For data members or member functions
  - `static` members: members that all objects of that class share
    - For data members or member functions
  - `mutable` data members : members that need no protection from changing any time but are encapsulated in class
    - For only data members

`const` (Constant) Objects

- Keyword `const`
- Specifies that an object is not modifiable
- Attempts to modify the object will result in compilation errors
- Example:

```
const Time noon(12,0,0);
noon.setHour(12); // ERROR
```

- Only `const` member functions can be called for `const` objects

Declaring variables and objects `const` when appropriate can help compilers to improve performance

`const` (Constant) Member Functions

- Member functions declared `const` are not allowed to modify the object (its data members)
  - Compilers will report errors if they do
- `const` declarations are not allowed for constructors and destructors
- A function is specified as `const` both in its prototype and in its definition

time.h

```
int getHour() const;
```

time.cpp

```
int Time::getHour() const {
  return hour;
}
```

Using const Member Functions

- A const member function cannot call a non-const member function
  - A const member function must explicitly be declared as const
    - A member function does not modify an object is not sufficient

time.h

```
int getHour();
void displayHour() const;
```

time.cpp

```
int Time::displayHour() const {
  cout << ”Hour is " << getHour()
    << endl; //ERROR
}
```

Overloading `const` Member Functions

- A `const` member function can be overloaded with a non-`const` version
- The complier decides which version to use:
  - `const` objects call `const` version
  - `non-const` objects call `non-const` version

```
int Time::displayHour() const {
    ... // cannot modify data members
}
int Time::displayHour() {
    ... // can modify data members
}
```

const Data Members

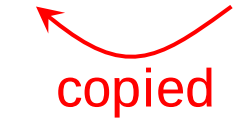
- const data members must be initialized with member initializer, so must data members declared as references
  - Assignments for const and reference data members in the constructor body are not allowed
- Member initializer list executes before the body of the constructor executes
  - Member initializers are separated by commas

classA.h

```
const int incrementStep;
```

classA.cpp

```
classA::classA():incrementStep(1) {
  ...
}
```



Increment.h

```
1  // Fig. 10.4: Increment.h
2  // Definition of class Increment.
3  #ifndef INCREMENT_H
4  #define INCREMENT_H
5
6  class Increment
7  {
8  public:
9     Increment( int c = 0, int i = 1 ); // default constructor
10
11    // function addIncrement definition
12    void addIncrement()
13    {
14       count += increment;
15    } // end function addIncrement
16
17    void print() const; // prints count and increment
18 private:
19    int count;
20    const int increment; // const data member
21 }; // end class Increment
22
23 #endif
```

Increment.cpp

Colon (:) marks the start of a member initializer list

```
1  // Fig. 10.5: Increment.cpp
2  // Member-function definitions for class Increment demonstrate using a
3  // member initializer to initialize a constant of a built-in data type.
4  #include <iostream>
5  #include "Increment.h" // include definition of class Increment
6  using namespace std;
7
8  // constructor
9  Increment::Increment( int c, int i )
10    : count( c ), // initializer for non-const member
11      increment( i ) // required initializer for const member
12 {
13    // empty body
14 } // end constructor Increment
15
16 // print count and increment values
17 void Increment::print() const
18 {
19    cout << "count = " << count << ", increment = " << increment << endl;
20 } // end function print
```

Member initializer for non-const member count

Required member initializer for const member increment

Erroneous Increment.cpp

```
1  // Fig. 10.8: Increment.cpp
2  // Erroneous attempt to initialize a constant of a built-in data
3  // type by assignment.
4  #include <iostream>
5  #include "Increment.h" // include definition of class Increment
6  using namespace std;
7
8  // constructor; constant member 'increment' is not initialized
9  Increment::Increment( int c, int i )
10 {
11    count = c; // allowed because count is not constant
12    increment = i; // ERROR: Cannot modify a const object
13 } // end constructor Increment
14
15 // print count and increment values
16 void Increment::print() const
17 {
18    cout << "count = " << count << ", increment = " << increment << endl;
19 } // end function print
```

Outline

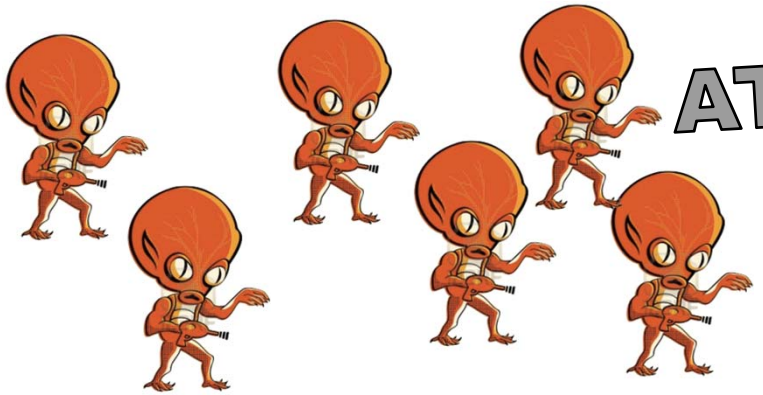
- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

`static` Data Members

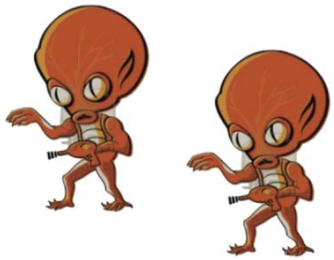
- Each object of a class has its own copy of all the data members of the class
  - Each object has its own attribute values
- In certain cases, only one copy of a variable should be shared by all objects of a class
  - A `static data member` is used for these reasons
  - "Class-wide" data members

An Example: Designing a Video Game

- If a Martian knows there are at least five Martians, each Martian is willing to attack other space creatures



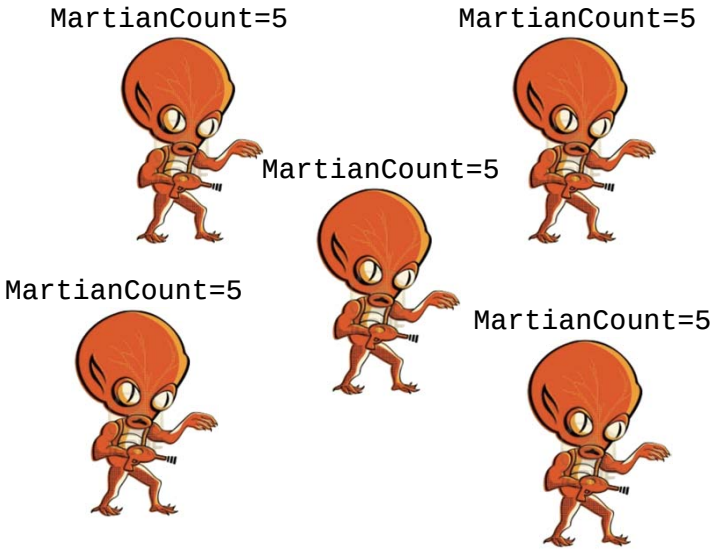
- Otherwise, each Martian becomes cowardly

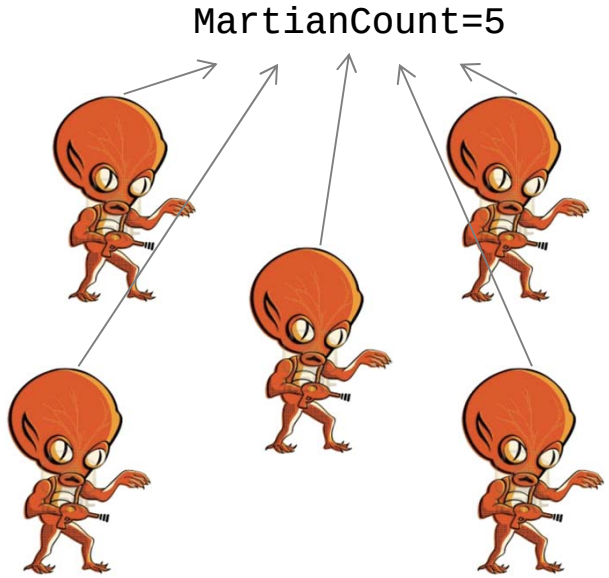


Two Ways to Design a `Martian` Class

```
class Martian {
   ...
   int MartianCount;
}
```

```
class Martian {
   ...
   static int MartianCount;
}
```





1. Less space required
2. Easier to maintain

Properties of `static` Data Members

- `static` data members have class scope
  - Although they may seem like global variables
- Can be `public`, `private`, or `protected`
- To access a `public static` class member when no objects of the class exist, simply prefix the class name and `::` to the name of the member
  - The member can be a data member or a member function

```
cout << Martian::MartianCount;
cout << Martian::getCount();
```

Initialization of `static` Data Members

- Initialized to 0 by default
- A `static const` data member of `int` or `enum` type can be initialized in its declaration in the class definition
- All other static data members must be defined (or initialized) at global namespace scope (i.e., outside the class body)
  - Declared (without initialization) inside the class (`.h`)
  - Initialized outside of the class (`.cpp`)
- If a static data member is an object of a class that provides a default constructor, it need not be initialized
  - The default constructor will be called

Initializing `static const int` Data Members

- Can be initialized in the interface

classA.h

```
class classA {
   static const int RATE = 0.27;
};
```

- Or initialized in the implementation

classA.h

```
class classA {
   static const int RATE;
};
```

No static keyword, why?

classA.cpp

```
const int classA::RATE = 0.27;
```

Employee.h

```
1  // Fig. 10.20: Employee.h
2  // Employee class definition with a static data member to
3  // track the number of Employee objects in memory
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8  using namespace std;
9
10 class Employee
11 {
12 public:
13    Employee( const string &, const string & ); // constructor
14    ~Employee(); // destructor
15    string getFirstName() const; // return first name
16    string getLastName() const; // return last name
17
18    // static member function
19    static int getCount(); // return number of objects instantiated
20 private:
21    string firstName;
22    string lastName;
23
24    // static data
25    static int count; // number of objects instantiated
26 }; // end class Employee
27
28 #endif
```

Employee.cpp

```
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  // define and initialize static data member at global namespace scope
8  int Employee::count = 0; // cannot include keyword static
9
10 // define static member function that returns number of
11 // Employee objects instantiated (declared static in Employee.h)
12 int Employee::getCount()
13 {
14    return count;
15 } // end static function getCount
16
17 // constructor initializes non-static data members and
18 // increments static data member count
19 Employee::Employee( const string &first, const string &last )
20    : firstName( first ), lastName( last )
21 {
22    ++count; // increment static count of employees
23    cout << "Employee constructor for " << firstName
24       << ' ' << lastName << " called." << endl;
25 } // end Employee constructor
26
27 // destructor deallocates dynamically allocated memory
28 Employee::~Employee()
29 {
30    cout << "~Employee() called for " << firstName
31       << ' ' << lastName << endl;
32    --count; // decrement static count of employees
33 } // end ~Employee destructor
```

A Client of Class Employee

```
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  using namespace std;
6
7  int main()
8  {
9     // no objects exist; use class name and binary scope resolution
10    // operator to access static member function getCount
11    cout << "Number of employees before instantiation of any objects is "
12       << Employee::getCount() << endl; // use class name
13
14    // the following scope creates and destroy
15    // Employee objects before main terminates
16    {
17       Employee e1( "Susan", "Baker" );
18       Employee e2( "Robert", "Jones" );
19
20       // two objects exist; call static membe
21       // using the class name and the binary
22       cout << "Number of employees after objects are instantiated is "
23          << Employee::getCount();
24
25       cout << "\n\nEmployee 1: "
26          << e1.getFirstName() << " " << e1.getLastName()
27          << "\nEmployee 2: "
28          << e2.getFirstName() << " " << e2.getLastName() << "\n\n";
29    } // end nested scope in main
30
31    // no objects exist, so call static member function getCount again
32    // using the class name and the binary scope resolution operator
33    cout << "\nNumber of employees after objects are deleted is "
34       << Employee::getCount() << endl;
35 } // end main
```

```
Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker

Number of employees after objects are deleted is 0
```

static Member Functions

- A member function should be declared `static` if it does not access non-`static` data members or non-`static` member functions of the class
- There is no `static const` member function
  - `const` indicates that a function cannot modify the contents of the object in which it operates, but `static` member functions exist and operate independently of any objects of the class

Outline

- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

mutable Data Members (1/2)

- mutable means volatile (opposite to constant)
  - A const member function cannot modify any data member which may not need protection
- mutable makes data member always changeable, even in a const member function

```
class A {
public:
  mutable int count;
  void foo() const;
};
```

```
void A::foo() const {
  count++; // legal
}
```

`mutable` Data Members (2/2)

- Modifying a `mutable` data member of a `const` object is legal

```
class A {
public:
  mutable int foo;
  int bar;
};
```

```
int main() {
  const A a;
  a.foo = 10; // legal
  a.bar = 10; // illegal

}
```

Outline

- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

Composition: Objects as Members of Classes

- Composition (a form of software reusability)
  - Sometimes referred to as a *has-a relationship*
  - A class can have objects of other classes as data members
  - Example
    - An `AlarmClock` class includes a `Time` object as a data member
- Initializing member objects
  - Member initializers pass arguments from the object's constructor to member-object constructors
  - The **copy constructor** of the member-object class is invoked

An Example of Composition

- Suppose we have defined a `Date` class, we are defining a `Employee` class
- Class `Employee` can contain `Date` objects (`birthDate` and `hireDate`) as it data members

Date.h

```
1  // Fig. 10.10: Date.h
2  // Date class definition; Member functions defined in Date.cpp
3  #ifndef DATE_H
4  #define DATE_H
5
6  class Date
7  {
8  public:
9     static const int monthsPerYear = 12; // number of months in a year
10    Date( int = 1, int = 1, int = 1900 ); // default constructor
11    void print() const; // print date in month/day/year format
12    ~Date(); // provided to confirm destruction order
13 private:
14    int month; // 1-12 (January-December)
15    int day; // 1-31 based on month
16    int year; // any year
17
18    // utility function to check if day is proper for month and year
19    int checkDay( int ) const;
20 }; // end class Date
21
22 #endif
```

Date.cpp (1/3)

```
1  // Fig. 10.11: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include "Date.h" // include Date class definition
5  using namespace std;
6
7  // constructor confirms proper value for month; calls
8  // utility function checkDay to confirm proper value for day
9  Date::Date( int mn, int dy, int yr )
10 {
11    if ( mn > 0 && mn <= monthsPerYear ) // validate the month
12       month = mn;
13    else
14    {
15       month = 1; // invalid month set to 1
16       cout << "Invalid month (" << mn << ") set to 1.\n";
17    } // end else
18
19    year = yr; // could validate yr
20    day = checkDay( dy ); // validate the day
21
22    // output Date object to show when its constructor is called
23    cout << "Date object constructor for date ";
```

Date.cpp (2/3)

```
24       print();
25       cout << endl;
26    } // end Date constructor
27
28    // print Date object in form month/day/year
29    void Date::print() const
30    {
31       cout << month << '/' << day << '/' << year;
32    } // end function print
33
34    // output Date object to show when its destructor is called
35    Date::~Date()
36    {
37       cout << "Date object destructor for date ";
38       print();
39       cout << endl;
40    } // end ~Date destructor
41
42    // utility function to confirm proper day value based on
43    // month and year; handles leap years, too
44    int Date::checkDay( int testDay ) const
45    {
46       static const int daysPerMonth[ monthsPerYear + 1 ] =
47          { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Date.cpp (3/3)

```
48
49       // determine whether testDay is valid for specified month
50       if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
51          return testDay;
52
53       // February 29 check for leap year
54       if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
55          ( year % 4 == 0 && year % 100 != 0 ) ) )
56          return testDay;
57
58       cout << "Invalid day (" << testDay << ") set to 1.\n";
59       return 1; // leave object in consistent state if bad value
60    } // end function checkDay
```

Employee.h

```
1  // Fig. 10.12: Employee.h
2  // Employee class definition showing composition.
3  // Member functions defined in Employee.cpp.
4  #ifndef EMPLOYEE_H
5  #define EMPLOYEE_H
6
7  #include <string>
8  #include "Date.h" // include Date class definition
9  using namespace std;
10
11 class Employee
12 {
13 public:
14    Employee( const string &, const string &,
15       const Date &, const Date & );
16    void print() const;
17    ~Employee(); // provided to confirm destruction order
18 private:
19    string firstName; // composition: member object
20    string lastName; // composition: member object
21    const Date birthDate; // composition: member object
22    const Date hireDate; // composition: member object
23 }; // end class Employee
24
25 #endif
```

Employee.cpp (1/2)

```
1  // Fig. 10.13: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  #include "Date.h" // Date class definition
6  using namespace std;
7
8  // constructor uses member initializer list to pass initializer
9  // values to constructors of member objects
10 Employee::Employee( const string &first, const string &last,
11    const Date &dateOfBirth, const Date &dateOfHire )
12    : firstName( first ), // initialize firstName
13      lastName( last ), // initialize lastName
14      birthDate( dateOfBirth ), // initialize birthDate
15      hireDate( dateOfHire ) // initialize hireDate
16 {
17    // output Employee object to show when constructor is called
18    cout << "Employee object constructor: "
19       << firstName << ' ' << lastName << endl;
20 } // end Employee constructor
21
```

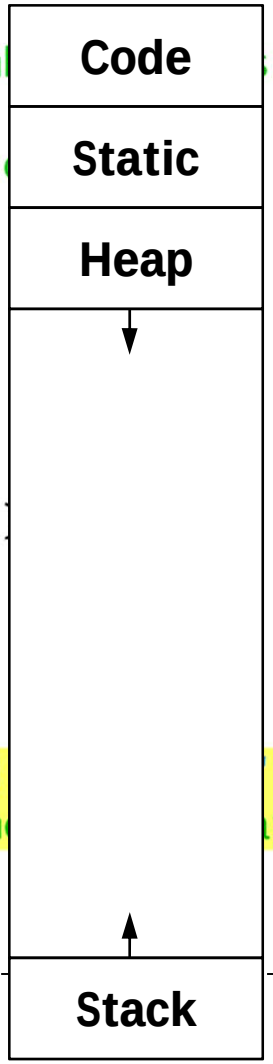
Copied by copy constructor

Employee.cpp (2/2)

```
22  // print Employee object
23  void Employee::print() const
24  {
25     cout << lastName << ", " << firstName << "  Hired: ";
26     hireDate.print();
27     cout << "  Birthday: ";
28     birthDate.print();
29     cout << endl;
30  } // end function print
31
32  // output Employee object to show when its destructor is called
33  Employee::~Employee()
34  {
35     cout << "Employee object destructor: "
36        << lastName << ", " << firstName << endl;
37  } // end ~Employee destructor
```

Using Classes Date and Employee

```
1  // Fig. 10.14: fig10_14.cpp
2  // Demonstrating composition--an object with meml
3  #include <iostream>
4  #include "Employee.h" // Employee class definiti
5  using namespace std;
6
7  int main()
8  {
9     Date birth( 7, 24, 1949 );
10    Date hire( 3, 12, 1988 );
11    Employee manager( "Bob", "Blue", birth, hire
12
13    cout << endl;
14    manager.print();
15
16    cout << "\nTest Date constructor with invalid
17    Date lastDayOff( 14, 35, 1994 ); // invalid m
18    cout << endl;
19 } // end main
```



Using Classes Date and Employee

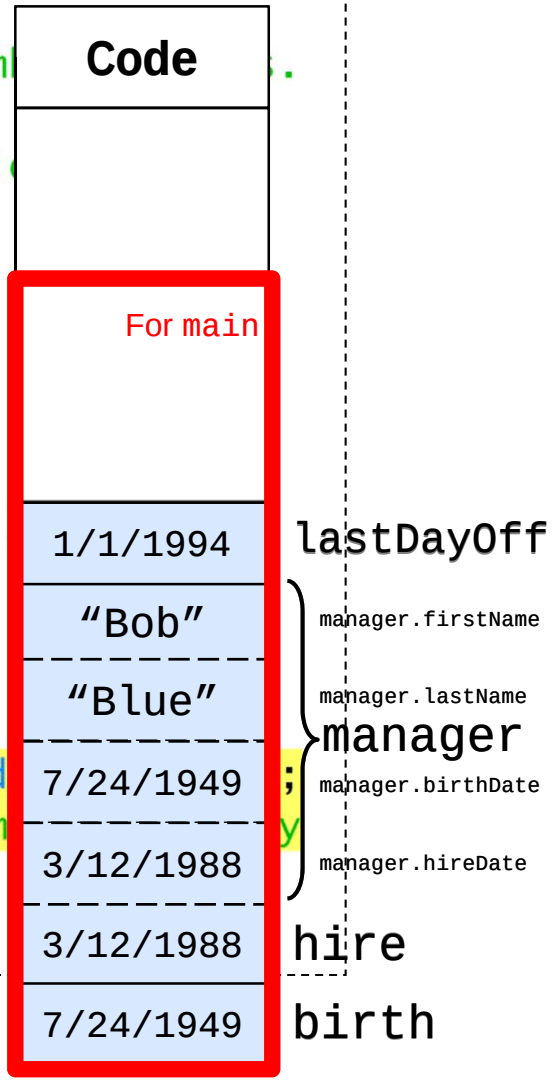
1. Pushing main's ARI to stack
2. Calling constructor for birth
3. Calling constructor for hire
4. Creating Employee object manager
   (1) Calling copy constructor for manager.firstName
   (2) Calling copy constructor for manager.lastName
   (3) Calling copy constructor for manager.birthDate
   (4) Calling copy constructor for manager.hireDate
5. Calling constructor for manager
6. Calling manager.print()
7. Calling constructor for lastDayOff
8. Calling destructor for lastDayOff
9. Calling destructor for manager
   (1) Calling destructor for manager.hireDate
   (2) Calling destructor for manager.birthDate
   (3) Calling destructor for manager.lastName
   (4) Calling destructor for manager.firstName
10. Calling destructor for hire
11. Calling destructor for birth
12. Popping main's ARI from stack

Depends on the order of data members being defined

Constructed from **inside out**

Destructed from **outside in**

```
1  int main() {
2    Employee manager(birth, hire);
3    manager.print();
4    lastDayOff = Date(1, 1, 1994);
5    manager = Employee("Bob", "Blue", birth, hire);
6    hire = Date(3, 12, 1988);
7    birth = Date(7, 24, 1949);
8    return 0;
9  } // end main
```



Using Classes `Date` and `Employee`

```
Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

Blue, Bob  Hired: 3/12/1988  Birthday: 7/24/1949

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
```

There are actually five constructor calls when an **`Employee`** is constructed—two calls to the **`string`** class's constructor (lines 12–13 of Fig. 10.13), two calls to the **`Date`** class's default copy constructor (lines 14–15 of Fig. 10.13) and the call to the **`Employee`** class's constructor.

Initializing Member Objects without Member Initializer

- A member object does not need to be initialized explicitly through a member initializer
- If a member object is not initialized with a member initializer, the member object's default constructor (with no arguments) will be called implicitly
  - If the member object's class does not provide a default constructor, a compilation error occurs
    - The member object's class defines one or more constructors, but none is a default constructor
- You can **re-initialize** a member object in the constructor of the enclosing class

Double Initialization

```
1  // Fig. 10.13: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
5  #include "Date.h" // Date class definition
6  using
7
8  // constructor us                    ializer first to pass initializer
9  // values to constructors o member objects
10 Employee::Employee( const string &first, const string &last,
11    const Date &dateOfBirth, const Date &dateOfHire )
12    : firstName( first ), // initialize firstName
13      lastName( last ), // initialize lastName
14      birthDate( dateOfBirth ), // initialize birthDate
15      hireDate( dateOfHire ) //
16 {
17    // output Employee object                    is called
18    cout << "Employee object constructor:
19       << firstName << ' ' << lastName <<
20 } // end Emp
21
```

Member initializer is not used for `firstName` and `lastName`, so the default constructor `string()` is called

```
firstName = first;
lastName = last;
```

`firstName` and `lastName` are initialized again by the memberwise assignment operator

Outline

- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

Nested Classes

- You can define a class within a class
  - Used when the inner (nested) class is meaningful inside the outer class
- A nested class can be either `public` or `private`
  - If a nested class is `private`, it cannot be used outside of the outer class
  - Whether the nested class is `public` or `private`, it can be used in the outer class
- A nested class cannot access any `private` member of the outer class by default

Nested Classes: An Example

```
class OuterClass {
public:
  class InnerClass1 {
    ...
  };
private:
  class InnerClass2 {
    ...
  };
  InnerClass1 obj1;
  InnerClass2 obj2;
};
```

```
int main() {
  OuterClass::InnerClass1 obj1;

  OuterClass::InnerClass2 obj2;

  return 0;
}
```

Local Classes

- You can also define a class within a function definition
  - The local class is confined to the function definition
  - A local class cannot contain `static` members

```
void function() {
  class LocalClass {
    ...
  };
  LocalClass obj;
};
```

Outline

- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

`friend` Classes (1/2)

- If class A is a friend of class B
  - A is granted to access the non-public and public members of B

```
class A {
  friend class B;
};
```

```
class B {
  int foo;
};
```

- The friendship is neither symmetric or transitive
  - If A is a friend of B, and B is a friend of C, you **cannot** infer that
    - B is a friend of A,
    - C is a friend of B, or
    - A is a friend of C

`friend` Classes (2/2)

- `friend` declarations can be placed anywhere in a class definition
  - They are not relevant to member access notations of `private`, `protected`, and `public`

An Example of `friend` Class (1/2)

CPoint.h

```
class CPoint {
  friend class CLine;
private:
  int x, y;
  void Offset(int diff) {
    x += diff; y += diff;
  }
public:
  CPoint() { x=0; y=0; }
  CPoint(int a, int b) { x=a; y=b; }
  void set(int a, int b) { x=a; y=b; }
  void Print() {
    cout << x << " " << y << endl;
  }
};
```

An Example of `friend` Class (2/2)

CLine.h

```
#include "CPoint.h"
class CLine {
private:
  CPoint p1, p2;
public:
  CLine(int x, int y, int w, int z) {
    p1.x = x; p1.y = y; // access private members
    p2.x = w; p2.y = z;
  }
  void Print() {
    cout << "Point 1:"; p1.Print();
    cout << "Point 2:"; p2.Print();
  }
  void Display() {
    p2.Offset(200); // call private function
    Print();
  }
};
```

`friend` Functions (1/2)

- 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
  - The prototype of the `friend` function must be placed in that class and preceded by reserved word `friend`
- `friend` functions are global functions with at least one argument that must be a class object or reference
  - `friend` functions are called using C-style function call
  - `friend` functions are not member functions
- A function can be a friend of two or more different classes

`friend` Functions (2/2)

- `friend` declarations can be placed anywhere in a class definition
  - They are not relevant to member access notations of `private`, `protected`, and `public`
- A friend function cannot be *inherited* (covered in Chapter 12, Inheritance)
- Necessary when you overload input and output operators for a class (will be discussed in Chapter 11, Operator Overloading)

Example of `friend` Function (1/2)

CPoint.h

```
class CPoint {
  friend void F_offset(CPoint &, int);
  friend class CLine;
private:
  int x, y;
  ...
};
void F_offset(CPoint &pt, int diff) {
  pt.x += diff; pt.y += diff;
}
```

```
#include "Cpoint.h"
int main() {
  CPoint p1;
  F_offset(p1, 4); // call friend function
}
```

Example of `friend` Function (2/2)

CLine.h

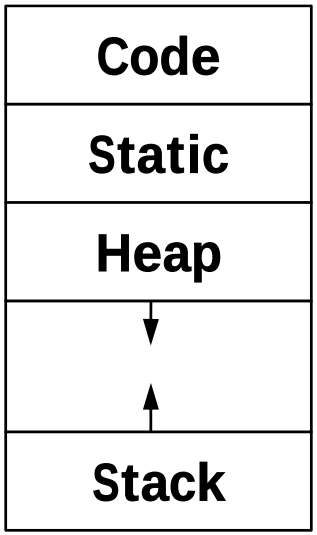
```
#include "CPoint.h"
class CLine {
  friend void F_offset(Cpoint &, int);
private:
  CPoint p1, p2;
public:
  ...
  void Display() {
    p2.Offset(200); // call private function
    F_offset(p1, 200); // call friend function
    Print();
  }
};
```

Outline

- Categories of Class Members
  - `const` Objects, `const` Member Functions, and `const` Data Members
  - `static` Data Members and `static` Member Functions
  - `mutable` Data Members
- Composition: Objects as Members of Classes
  - Nested and Local Classes
- `friend` Functions and `friend` Classes
- `this`: A Special Pointer

Using the `this` Pointer (1/3)

- It is a common knowledge that C++ keeps only one copy of each member function
- 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++ reserved word)



Using the `this` Pointer (2/3)

- The `this` pointer is passed (by the compiler) as an implicit argument to each of the object's non-static member functions
- A `static` member function does not have a `this` pointer
- The `this` pointer is not counted for calculating the size of the object (`sizeof`)

Using the `this` Pointer (3/3)

- Member functions use the `this` pointer implicitly or explicitly
  - Implicitly when accessing members directly
  - Explicitly when using keyword `this`
  - Example:

```
void Time::setHour(int h) {  //implicitly
  hour = h;
}
```

```
void Time::setHour(int hour) { //explicitly
  this->hour = hour;
}
```

Using the `this` Pointer: An Example

```
1  // Fig. 10.16: fig10_16.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using namespace std;
5
6  class Test
7  {
8  public:
9     Test( int = 0 ); // default constructor
10    void print() const;
11 private:
12    int x;
13 }; // end class Test
14
15 // constructor
16 Test::Test( int value )
17    : x( value ) // initia
18 {
19    // empty body
20 } // end constructor Test
```

```
22 // print x using implicit and explicit this pointers;
23 // the parentheses around *this are required
24 void Test::print() const
25 {
26    // implicitly use the this pointer to access the member x
27    cout << "        x = " << x;
28
29    // explicitly use the this pointer and the arrow operator
30    // to access the member x
31    cout << "\n  this->x = " << this->x;
32
33    // explicitly use the dereferenced this pointer and
34    // the dot operator to access the member x
35    cout << "\n(*this).x = " << ( *this ).x << endl;
36 } // end function print
37
38 int main()
39 {
40    Test testObject( 12 ); // instantiate and initialize testObject
41
42    testObject.print();
43 } // end main
```

Enabling Cascaded Member-Function Calls

- Another use of the `this` pointer is to enable cascaded member-function calls
  - invoking multiple functions in the same statement
  - `object.foo().bar();`

Time.h

```
 6  #ifndef TIME_H
 7  #define TIME_H
 8
 9  class Time
10  {
11  public:
12     Time( int = 0, int = 0, int = 0 ); // default constructor
13
14     // set functions (the Time & return types enable cascading)
15     Time &setTime( int, int, int ); // set hour, minute, second
16     Time &setHour( int ); // set hour
17     Time &setMinute( int ); // set minute
18     Time &setSecond( int ); // set second
19
20     // get functions (normally declared const)
21     int getHour() const; // return hour
22     int getMinute() const; // return minute
23     int getSecond() const; // return second
24
25     // print functions (normally declared const)
26     void printUniversal() const; // print universal time
27     void printStandard() const; // print standard time
28  private:
29     int hour; // 0 - 23 (24-hour clock format)
30     int minute; // 0 - 59
31     int second; // 0 - 59
32  }; // end class Time
33
34  #endif
```

Time.cpp

```
1  // Fig. 10.18: Time.cpp
2  // Time class member-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include "Time.h" // Time class definition
6  using namespace std;
7
8  // constructor function to initialize private data;
9  // calls member function setTime to set variables;
10 // default values are 0 (see class definition)
11 Time::Time( int hr, int min, int sec )
12 {
13    setTime( hr, min, sec );
14 } // end Time constructor
15
16 // set values of hour, minute, and second
17 Time &Time::setTime( int h, int m, int s ) // note Time & return
18 {
19    setHour( h );
20    setMinute( m );
21    setSecond( s );
22    return *this; // enables cascading
23 } // end function setTime
```

A Client of Class Time

```
 3  #include <iostream>
 4  #include "Time.h" // Time class definition
 5  using namespace std;
 6
 7  int main()
 8  {
 9     Time t; // create Time object
10
11     // cascaded function calls
12     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
13
14     // output time in universal and standard formats
15     cout << "Universal time: ";
16     t.printUniversal();
17
18     cout << "\nStandard time: ";
19     t.printStandard();
20
21     cout << "\n\nNew standard time: ";
22
23     // cascaded function calls
24     t.setTime( 20, 20, 20 ).printStandard();
25     cout << endl;
26  } // end main
```

```
Universal time: 18:30:22
Standard time: 6:30:22 PM

New standard time: 8:20:20 PM
```