# Classes: A Deeper Look

# Deeper into C++ Classes

- const objects and const member functions
- Composition
- Friendship
- this pointer
- Dynamic memory management
  - new and delete operators
- static class members and member functions

# 18.2 const (Constant) Objects and const Member Functions

- Principle of least privilege
  - "allowing access to data only when it is absolutely needed."
  - Is one of the most fundamental principles of good software engineering.
  - Applies to objects, too.
- const 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);

# const (Constant) Objects and const Member Functions

- const member functions
  - Only const member function can be called for const objects.
  - Member functions declared const are not allowed to modify the object.
  - A function is specified as const both in its prototype and in its definition.
  - const declarations are not allowed for constructors and destructors.

# Software Engineering Observation 18.2

- 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.

# const Example

```
1  // Fig. 21.1: Time.h
2  // Definition of class Time.
3  // Member functions defined in Time.cpp.
4  #ifndef TIME_H
5  #define TIME_H
6
7  class Time
8  {
9  public:
10    Time( int = 0, int = 0, int = 0 ); // default constructor
11
12    // set functions
13    void setTime( int, int, int ); // set time
14    void setHour( int ); // set hour
15    void setMinute( int ); // set minute
16    void setSecond( int ); // set second
17
18    // get functions (normally declared const)
19    int getHour() const; // return hour
20    int getMinute() const; // return minute
21    int getSecond() const; // return second
```

# const Example

```
22
23    // print functions (normally declared const)
24    void printUniversal() const; // print universal time
25    void printStandard(); // print standard time (should be const)
26 private:
27    int hour; // 0 - 23 (24-hour clock format)
28    int minute; // 0 - 59
29    int second; // 0 - 59
30 }; // end class Time
31
32 #endif
```

# const Example

```
1  // Fig. 21.2: Time.cpp
2  // Member-function definitions for class Time.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // include definition of class Time
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hour, int minute, int second )
16 {
17    setTime( hour, minute, second );
18 } // end Time constructor
19
20 // set hour, minute and second values
21 void Time::setTime( int hour, int minute, int second )
22 {
23    setHour( hour );
24    setMinute( minute );
25    setSecond( second );
26 } // end function setTime
```

# const Example

```
27
28 // set hour value
29 void Time::setHour( int h )
30 {
31    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
32 } // end function setHour
33
34 // set minute value
35 void Time::setMinute( int m )
36 {
37    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
38 } // end function setMinute
39
40 // set second value
41 void Time::setSecond( int s )
42 {
43    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
44 } // end function setSecond
45
46 // return hour value
47 int Time::getHour() const // get functions should be const
48 {
49    return hour;
50 } // end function getHour
```

const keyword in function definition, as well as in function prototype

# const Example

```
51
52 // return minute value
53 int Time::getMinute() const
54 {
55    return minute;
56 } // end function getMinute
57
58 // return second value
59 int Time::getSecond() const
60 {
61    return second;
62 } // end function getSecond
63
64 // print Time in universal-time format (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // end function printUniversal
70
71 // print Time in standard-time format (HH:MM:SS AM or PM)
72 void Time::printStandard() // note lack of const declaration
73 {
74    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75       << ":" << setfill( '0' ) << setw( 2 ) << minute
76       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // end function printStandard
```

# const Example

```
1  // Fig. 21.3: fig21_03.cpp
2  // Attempting to access a const object with non-const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main()
6  {
7     Time wakeUp( 6, 45, 0 ); // non-constant object
8     const Time noon( 12, 0, 0 ); // constant object
9
10                              // OBJECT      MEMBER FUNCTION
11    wakeUp.setHour( 18 );     // non-const   non-const
12
13    noon.setHour( 12 );       // const       non-const
14
15    wakeUp.getHour();         // non-const   const
16
17    noon.getMinute();         // const       const
18    noon.printUniversal();    // const       const
19
20    noon.printStandard();     // const       non-const
21    return 0;
22 } // end main
```

Cannot invoke `non-const` member functions on a `const` object

# const Example

*Borland C++ command-line compiler error messages:*

```
Warning W8037 fig21_03.cpp 13: Non-const function Time::setHour(int)
   called for const object in function main()
Warning W8037 fig21_03.cpp 20: Non-const function Time::printStandard()
   called for const object in function main()
```

*Microsoft Visual C++.NET compiler error messages:*

```
C:\examples\ch21\Fig21_01_03\fig21_03.cpp(13) : error C2662:
   'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
C:\examples\ch21\Fig21_01_03\fig21_03.cpp(20) : error C2662:
   'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
   'Time &'
        Conversion loses qualifiers
```

*GNU C++ compiler error messages:*

```
Fig21_03.cpp:13: error: passing `const Time' as `this' argument of
   `void Time::setHour(int)' discards qualifiers
Fig21_03.cpp:20: error: passing `const Time' as `this' argument of
   `void Time::printStandard()' discards qualifiers
```

# Member Initializer

- Required for initializing:
  - const data members
  - data members that are references.
- Can be used for any data member.
- Member initializer list
  - Appears 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.
  - Multiple member initializers are separated by commas.
  - Executes before the body of the constructor executes.

# Member Initializer

```
1  // Fig. 21.4: 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
```

**const data member that must be initialized using a member initializer**

# Member Initializer

```
1  // Fig. 21.5: 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  using std::cout;
6  using std::endl;
7
8  #include "Increment.h" // include definition of class Increment
9
10 // constructor
11 Increment::Increment( int c, int i )
12    : count( c ), // initializer for non-const member
13      increment( i ) // required initializer for const member
14 {
15    // empty body
16 } // end constructor Increment
17
18 // print count and increment values
19 void Increment::print() const
20 {
21    cout << "count = " << count << ", increment = " << increment << endl;
22 } // end function print
```

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

Member initializer for non-const member count

Required member initializer for const member increment

print being const member function not required, but safer

# Member Initializer

```
1  // Fig. 21.6: fig21_06.cpp
2  // Program to test class Increment.
3  #include <iostream>
4  using std::cout;
5
6  #include "Increment.h" // include definition of class Increment
7
8  int main()
9  {
10    Increment value( 10, 5 );
11
12    cout << "Before incrementing: ";
13    value.print();
14
15    for ( int j = 1; j <= 3; j++ )
16    {
17       value.addIncrement();
18       cout << "After increment " << j << ": ";
19       value.print();
20    } // end for
21
22    return 0;
23 } // end main
```

```
Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5
```

# Software Engineering Observation 18.3

- A const object cannot be modified by assignment, so it must be initialized. When a data member of a class is declared const, a member initializer must be used to provide the constructor with the initial value of the data member for an object of the class. The same is true for references.

# Common Programming Error 18.5

- Not providing a member initializer for a **const** data member is a compilation error.

# Error-Prevention Tip 18.1

**Error-Prevention Tip 10.1**

Declare as `const` all of a class's member functions that do not modify the object in which they operate. Occasionally this may seem inappropriate, because you'll have no intention of creating `const` objects of that class or accessing objects of that class through `const` references or pointers to `const`. Declaring such member functions `const` does offer a benefit, though. If the member function is inadvertently written to modify the object, the compiler will issue an error message.

# 18.3 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.
  - Example
    - **AlarmClock** object with a **Time** object as a member.

# Composition: Objects as Members of Classes

- Initializing member objects
  - Member initializers pass arguments from the object's constructor to member-object constructors.
  - Member objects are constructed in the order in which they are declared in the class definition.
    - Not in the order that they are listed in the constructor's member initializer list.
    - Before the enclosing class object (host object) is constructed.
  - If a member initializer is not provided
    - The member object's default constructor will be called implicitly.

# Software Engineering Observation 18.4

A common form of software reusability is **composition**, in which a class has objects of other classes as members.

# 18.8 Composition Example

```
1  // Fig. 10.8: 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
```

First example of static data member in C++

Fig. 10.8 | Date class definition.

# 18.9 Composition Example

```
1  // Fig. 10.9: Date.cpp
2  // Date class member-function definitions.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Date.h" // include Date class definition
6  using namespace std;
7
8  // constructor confirms proper value for month; calls
9  // utility function checkDay to confirm proper value for day
10 Date::Date( int mn, int dy, int yr )
11 {
12    if ( mn > 0 && mn <= monthsPerYear ) // validate the month
13       month = mn;
14    else
15       throw invalid_argument( "month must be 1-12" );
16
17    year = yr; // could validate yr
18    day = checkDay( dy ); // validate the day
19
20    // output Date object to show when its constructor is called
21    cout << "Date object constructor for date ";
22    print();
23    cout << endl;
24 } // end Date constructor
```

**Fig. 10.9** | Date class member-function definitions. (Part 1 of 3.)

# 18.9 Composition Example

```
25
26  // print Date object in form month/day/year
27  void Date::print() const
28  {
29     cout << month << '/' << day << '/' << year;
30  } // end function print
31
32  // output Date object to show when its destructor is called
33  Date::~Date()
34  {
35     cout << "Date object destructor for date ";
36     print();
37     cout << endl;
38  } // end ~Date destructor
39
```

**Fig. 10.9** | Date class member-function definitions. (Part 2 of 3.)

# 18.9 Composition Example

```
40  // utility function to confirm proper day value based on
41  // month and year; handles leap years, too
42  int Date::checkDay( int testDay ) const
43  {
44     static const int daysPerMonth[ monthsPerYear + 1 ] =
45        { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
46
47     // determine whether testDay is valid for specified month
48     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
49        return testDay;
50
51     // February 29 check for leap year
52     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
53        ( year % 4 == 0 && year % 100 != 0 ) ) )
54        return testDay;
55
56     throw invalid_argument( "Invalid day for current month and year" );
57  } // end function checkDay
```

Standard C trick

Fig. 10.9 | Date class member-function definitions. (Part 3 of 3.)

# 18.10 Composition Example

```
1  // Fig. 10.10: 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
```

**Fig. 10.10** | Employee class definition showing composition. (Part 1 of 2.)

# 18.10 Composition Example

```
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 destruc
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
```

**Parameters to be passed via member initializers to the constructor for class `Date`**

**`const` objects of class `Date` as members**

**utilizes a default copy constructor**

**Fig. 10.10** | Employee class definition showing composition. (Part 2 of 2.)

# 18. 11 Composition Example

```
1  // Fig. 10.11: Employee.cpp
2  // Employee class member-function definitions.
3  #include <iostream>
4  #include "Employee.h" // Employee class definiti
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
```

**Member initializers that pass arguments to `Date`'s implicit default copy constructor**

**Fig. 10.11** | Employee class member-function definitions, including constructor with a member initializer list. (Part 1 of 2.)

# 18.11 Composition Example

```
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
```

**Fig. 10.11** | Employee class member-function definitions, including constructor with a member initializer list. (Part 2 of 2.)

# 18.3 Composition: Objects as Members of Classes (cont.)

- As you study class Date (Fig. 18.8), notice that the class does not provide a constructor that receives a parameter of type Date.
- Why can the Employee constructor's member initializer list initialize the birthDate and hireDate objects by passing Date object's to their Date 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.

# 18.12 Composition Example

```
1  // Fig. 10.12: fig10_12.cpp
2  // Demonstrating composition--an object with member objects.
3  #include <iostream>
4  #include "Employee.h" // Employee class definition
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 } // end main
```

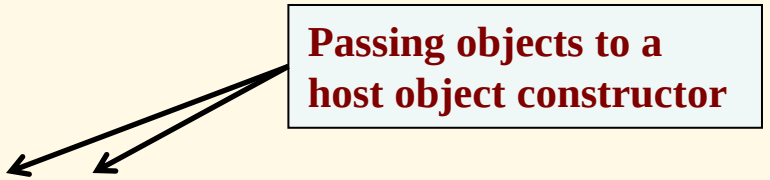


**Fig. 10.12** | Demonstrating composition—an object with member objects. (Part 1 of 2.)

# 18.12 Composition Example

```
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
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. 10.11), two calls to the `Date` class's default copy constructor (lines 14–15 of

**Fig. 10.12** | Demonstrating composition—an object with member objects. (Part 2 of 2.)

# Common Programming Error 18.6

- A compilation error occurs if a member object is not initialized with a member initializer and the member object's class does not provide a default constructor (i.e., the member object's class defines one or more constructors, but none is a default constructor).

# 18.3 Composition: Objects as Members of Classes (cont.)

- If a member object is not initialized through a member initializer, the member object's default constructor will be called implicitly.
- Values, if any, established by the default constructor can be overridden by set functions.
- However, for complex initialization, this approach may require significant additional work and time.

# 18.4 friend Functions and friend Classes

- friend function of a class
  - Defined outside that class's scope.
  - Not a member function of that class.
  - has the right to access the non-public and public members of that class.
  - Standalone functions, entire classes or member functions of other classes may be declared to be friends of a class.
  - Using friend can enhance performance.
  - Often appropriate when a member function cannot be used for certain operations.

# friend Functions and friend Classes

- To declare a function as a **friend** of a class:
  - Provide the function prototype in the class definition preceded by keyword **friend**.
- To declare a class as a friend of another class:
  - Place a declaration of the form

    **friend class ClassTwo;**

    in the definition of class **ClassOne**
- All member functions of class **ClassTwo** are **friends** of class **ClassOne**.

# friend Functions and friend Classes

- Friendship is granted, not taken.
  - For class B to be a friend of class A, class A must explicitly declare that class B is its friend.
- Friendship relation is neither symmetric nor transitive
  - If class A is a friend of class B, and class B is a friend of class C, you cannot infer that class B is a friend of class A, that class C is a friend of class B, or that class A is a friend of class C.

# friend Functions and friend Classes

- It is possible to specify overloaded functions as friends of a class.
  - Each overloaded function intended to be a friend must be explicitly declared as a friend of the class.

# Fig 18.13 friend Function Example

```
1  // Fig. 21.15: fig21_15.cpp
2  // Friends can access private members of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition
8  class Count
9  {
10    friend void setX( Count &, int ); // friend declaration
11 public:
12    // constructor
13    Count()
14       : x( 0 ) // initialize x to 0
15    {
16       // empty body
17    } // end constructor Count
18
19    // output x
20    void print() const
21    {
22       cout << x << endl;
23    } // end function print
24 private:
25    int x; // data member
26 }; // end class Count
```

**friend function declaration (can appear anywhere in the class)**

# Fig 18.13 friend Function Example

```
27
28 // function setX can modify private data of Count
29 // because setX is declared as a friend of Count (line 10)
30 void setX( Count &c, int val )
31 {
32    c.x = val; // allowed because setX is a friend of Count
33 } // end function setX
34
35 int main()
36 {
37    Count counter; // create Count object
38
39    cout << "counter.x after instantiation: ";
40    counter.print();
41
42    setX( counter, 8 ); // set x using a friend function
43    cout << "counter.x after call to setX friend function: ";
44    counter.print();
45    return 0;
46 } // end main
```

friend function can modify Count's private data

Calling a friend function; note that we pass the Count object to the function

```
counter.x after instantiation: 0
counter.x after call to setX friend function: 8
```

# non-friend Function Example

```
1  // Fig. 10.16: fig10_16.cpp
2  // Non-friend/non-member functions cannot access private data of a class.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  // Count class definition (note that there is no friendship declaration)
8  class Count
9  {
10 public:
11    // constructor
12    Count()
13       : x( 0 ) // initialize x to 0
14    {
15       // empty body
16    } // end constructor Count
17
18    // output x
19    void print() const
20    {
21       cout << x << endl;
22    } // end function print
23 private:
24    int x; // data member
25 }; // end class Count
```

# non-friend Function Example

```
26
27 // function cannotSetX tries to modify private data of Count,
28 // but cannot because the function is not a friend of Count
29 void cannotSetX( Count &c, int val )
30 {
31    c.x = val; // ERROR: cannot access private member in Count
32 } // end function cannotSetX
33
34 int main()
35 {
36    Count counter; // create Count object
37
38    cannotSetX( counter, 3 ); // cannotSetX is not a friend
39    return 0;
40 } // end main
```

Non-friend function cannot access the class's private data

# non-friend Function Example

*Borland C++ command-line compiler error message:*

```
Error E2247 Fig21_16/fig21_16.cpp 31: 'Count::x' is not accessible in
   function cannotSetX(Count &,int)
```

*Microsoft Visual C++.NET compiler error messages:*

```
C:\examples\ch21\Fig21_16\fig21_16.cpp(31) : error C2248: 'Count::x'
   : cannot access private member declared in class 'Count'
        C:\examples\ch21\Fig21_16\fig21_16.cpp(24) : see declaration
           of 'Count::x'
        C:\examples\ch21\Fig21_16\fig21_16.cpp(9) : see declaration
           of 'Count'
```

*GNU C++ compiler error messages:*

```
Fig21_16.cpp:24: error: 'int Count::x' is private
Fig21_16.cpp:31: error: within this context
```

# 18.5 Using the this Pointer

- 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).
  - An object's 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.

# 18.5 Using the this Pointer

- Objects use the this pointer implicitly or explicitly.
  - this is used implicitly when accessing members directly.
  - It is used explicitly when using keyword this.
  - The type of the this pointer depends on the type of the object and whether the executing member function is declared const.

# Fig 18.14 this Example

```
1  // Fig. 21.17: fig21_17.cpp
2  // Using the this pointer to refer to object members.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  class Test
8  {
9  public:
10    Test( int = 0 ); // default constructor
11    void print() const;
12 private:
13    int x;
14 }; // end class Test
15
16 // constructor
17 Test::Test( int value )
18    : x( value ) // initialize x to value
19 {
20    // empty body
21 } // end constructor Test
```

# Fig 18.14 this Example

```
22
23 // print x using implicit and explicit this pointers;
24 // the parentheses around *this are required
25 void Test::print() const
26 {
27    // implicitly use the this pointer to access the member x
28    cout << "        x = " << x;
29
30    // explicitly use the this pointer and the arrow operator
31    // to access the member x
32    cout << "\n  this->x = " << this->x;
33
34    // explicitly use the dereferenced this pointer and
35    // the dot operator to access the member x
36    cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // end function print
38
39 int main()
40 {
41    Test testObject( 12 ); // instantiate and initialize testObject
42
43    testObject.print();
44    return 0;
45 } // end main
```

Implicitly using the this pointer to access member x

Explicitly using the this pointer to access member x

Using the dereferenced this pointer and the dot operator

```
        x = 12
  this->x = 12
(*this).x = 12
```

# Common Programming Error 18.7

- Attempting to use the member selection operator (.) with a pointer to an object is a compilation error—the dot member selection operator may be used only with an lvalue such as an object's name, a reference to an object or a dereferenced pointer to an object.

# Using the this Pointer

- Cascaded member-function calls
  - Multiple functions are invoked in the same statement.
    - Enabled by member functions returning a reference to an object via the this pointer.
  - Example
    - t.setMinute( 30 ).setSecond( 22 );
      - Calls t.setMinute( 30 );
      - Then calls t.setSecond( 22 );

# Cascading Function Calls using the this Pointer

```
1  // Fig. 21.18: Time.h
2  // Cascading member function calls.
3
4  // Time class definition.
5  // Member functions defined in Time.cpp.
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
```

set functions return Time & (a reference) to enable cascading

# Cascading Function Calls using the this Pointer

```
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
```

# Cascading Function Calls using the this Pointer

```
1  // Fig. 21.19: Time.cpp
2  // Member-function definitions for Time class.
3  #include <iostream>
4  using std::cout;
5
6  #include <iomanip>
7  using std::setfill;
8  using std::setw;
9
10 #include "Time.h" // Time class definition
11
12 // constructor function to initialize private data;
13 // calls member function setTime to set variables;
14 // default values are 0 (see class definition)
15 Time::Time( int hr, int min, int sec )
16 {
17    setTime( hr, min, sec );
18 } // end Time constructor
19
20 // set values of hour, minute, and second
21 Time &Time::setTime( int h, int m, int s ) // note Time & return
22 {
23    setHour( h );
24    setMinute( m );
25    setSecond( s );
26    return *this; // enables cascading
27 } // end function setTime
```

Returning *this pointer enables cascading

# Cascading Function Calls using the this Pointer

```
28
29 // set hour value
30 Time &Time::setHour( int h ) // note Time & return
31 {
32    hour = ( h >= 0 && h < 24 ) ? h : 0; // validate hour
33    return *this; // enables cascading
34 } // end function setHour
35
36 // set minute value
37 Time &Time::setMinute( int m ) // note Time & return
38 {
39    minute = ( m >= 0 && m < 60 ) ? m : 0; // validate minute
40    return *this; // enables cascading
41 } // end function setMinute
42
43 // set second value
44 Time &Time::setSecond( int s ) // note Time & return
45 {
46    second = ( s >= 0 && s < 60 ) ? s : 0; // validate second
47    return *this; // enables cascading
48 } // end function setSecond
49
50 // get hour value
51 int Time::getHour() const
52 {
53    return hour;
54 } // end function getHour
```

# Cascading Function Calls using the this Pointer

```
55
56 // get minute value
57 int Time::getMinute() const
58 {
59    return minute;
60 } // end function getMinute
61
62 // get second value
63 int Time::getSecond() const
64 {
65    return second;
66 } // end function getSecond
67
68 // print Time in universal-time format (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71    cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72       << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // end function printUniversal
74
75 // print Time in standard-time format (HH:MM:SS AM or PM)
76 void Time::printStandard() const
77 {
78    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79       << ":" << setfill( '0' ) << setw( 2 ) << minute
80       << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // end function printStandard
```

# Cascading Function Calls using the this Pointer

```
1  // Fig. 21.20: fig21_20.cpp
2  // Cascading member function calls with the this pointer.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Time.h" // Time class definition
8
9  int main()
10 {
11    Time t; // create Time object
12
13    // cascaded function calls
14    t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
15
16    // output time in universal and standard formats
17    cout << "Universal time: ";
18    t.printUniversal();
19
20    cout << "\nStandard time: ";
21    t.printStandard();
22
23    cout << "\n\nNew standard time: ";
24
25    // cascaded function calls
26    t.setTime( 20, 20, 20 ).printStandard();
27    cout << endl;
28    return 0;
29 } // end main
```

Cascaded function calls using the reference returned by one function call to invoke the next

Note that these calls must appear in the order shown, because printStandard does not return a reference to t

# Cascading Function Calls using the this Pointer

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

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

# 19.9 Dynamic Memory Management: Operators new and delete

- Dynamic memory management in C++
  - Enables programmers to allocate and deallocate memory for objects, arrays or any built-in or user-defined type.
  - Performed by operators new and delete.
  - For example, dynamically allocating memory for an array instead of using a fixed-size array.

# 19.9 Operators new and delete

- Operator new
  - Allocates (i.e., reserves) storage of the exact for an object from the free store at execution time.
  - Calls a default constructor to initialize the object.
  - Returns a pointer of the type specified to the right of new (e.g., Time * below).
  - Can be used to dynamically allocate any fundamental type (such as int or double) or any class type.
- The free store (referred to as the heap)
  - Is a region of memory assigned to each program for storing objects created at execution time.

Example:

```
Time *timePtr
timePtr = new Time;
```

# 19.9 Operators new and delete

- Operator delete
  - Destroys a dynamically allocated object.
  - Calls the destructor for the object (e.g. to which timePtr points below).
  - Deallocates (i.e., releases) memory from the free store.
  - The memory can then be reused by the system to allocate other objects.

Example:

```
delete timePtr;
```

# 19.9 Operators new and delete

- Initializing an object allocated by new
  - Initializer for a newly created fundamental-type variable.

Example

```
double *ptr = new double( 3.14159 );
```

  - Specify a comma-separated list of arguments to the constructor of an object.

Example

```
Time *timePtr = new Time( 12, 45, 0 );
```

# 19.9 Operators new and delete

- new operator can be used to allocate arrays dynamically.
  - Dynamically allocate a 10-element integer array:

    ```
    int *gradesArray = new int[ 10 ];
    ```

  - Size of a dynamically allocated array
    - Specified using any integral expression that can be evaluated at execution time.

    ```
    Queue * queuePtr = new Queue[mules];
    ```

# 19.9 Operators new and delete

- Delete a dynamically allocated array:

  **delete [] gradesArray;**

  - This deallocates the array to which **gradesArray** points.
  - If the pointer points to an array of objects,
    - It first calls the destructor for every object in the array.
    - Then it deallocates the memory.
  - If the statement did not include the square brackets ([]) and **gradesArray** pointed to an array of objects : result is undefined!!
    - Some compilers would call destructor for only the first object in the array.

# 18.6 static Class Members

- static data member
  - When only one copy of a variable is shared by all objects of a class.
    - The member is "class-wide" information.
    - A property of the class shared by all instances, not a property of a specific object of the class.
  - Static data members can save storage.
  - Declaration begins with keyword static.

# 18.6 static Class Members

- IMGD Example
  - Video game with Martians and other space creatures
    - Each Martian needs to know the martianCount.
    - martianCount should be static class-wide data.
    - Every Martian can access martianCount as if it were a data member of that Martian
    - Only one copy of martianCount exists.
  - May seem like global variables but static data members have class scope.
  - Can be declared public, private or protected.

# 18.6 static Class Members

- Fundamental-type static data members
  - Initialized by default to 0.
  - If you want a different initial value, a static data member can be initialized once (and only once).
- 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 at file scope (i.e., outside the body of the class definition).
  - Can be initialized only in those definitions.
- static data members of class types (i.e., static member objects) that have default constructors
  - Need not be initialized because their default constructors will be called.

# 18.6 static Class Members

- Exists even when no objects of the 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.
      - Example

        Martian::martianCount
  - Also accessible through any object of that class
    - Use the object's name, the dot operator and the name of the member.
      - Example

        myMartian.martianCount

# 18.6 static Class Members

- static member function
  - Is a service of the class, not of a specific object of the class.
- static is applied to an item at file scope.
  - That item becomes known only in that file.
  - The static members of the class need to be available from any client code that accesses the file.
    - So we cannot declare them static in the .cpp file—we declare them static only in the .h file.

# Presentation Note

- The following example is older and from the 5th Edition of the Deitel textbook.
- This example is more complicated (but useful) because it provides an example of pointers to member functions and explicit new and delete memory allocation and deallocation calls .
- For an easier example of static data and member functions, see Figures 18.18 to 18.20 in the 7th Edition of Deitel & Deitel.

# static class member Example

```
1  // Fig. 21.21: Employee.h
2  // Employee class definition.
3  #ifndef EMPLOYEE_H
4  #define EMPLOYEE_H
5
6  class Employee
7  {
8  public:
9     Employee( const char * const, const char * const ); // constructor
10    ~Employee(); // destructor
11    const char *getFirstName() const; // return first name
12    const char *getLastName() const; // return last name
13
14    // static member function
15    static int getCount(); // return number of objects instantiated
16 private:
17    char *firstName;
18    char *lastName;
19
20    // static data
21    static int count; // number of objects instantiated
22 }; // end class Employee
23
24 #endif
```

Function prototype for static member function

static data member keeps track of number of Employee objects that currently exist

# static class member Example

```
1  // Fig. 21.22: Employee.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // strlen and strcpy prototypes
8  using std::strlen;
9  using std::strcpy;
10
11 #include "Employee.h" // Employee class definition
12
13 // define and initialize static data member at file scope
14 int Employee::count = 0;
15
16 // define static member function that returns number of
17 // Employee objects instantiated (declared static in Employee.h)
18 int Employee::getCount()
19 {
20    return count;
21 } // end static function getCount
```

**static** data member is defined and initialized at global scope in the .cpp file. (NO static keyword here!)

**static** member function can access only **static** data, because the function might be called when no objects exist.

# static class member Example

```
22
23 // constructor dynamically allocates space for first and last name and
24 // uses strcpy to copy first and last names into the object
25 Employee::Employee( const char * const first, const char * const last )
26 {
27    firstName = new char[ strlen( first ) + 1 ];
28    strcpy( firstName, first );
29
30    lastName = new char[ strlen( last ) + 1 ];
31    strcpy( lastName, last );
32
33    count++; // increment static count of employees
34
35    cout << "Employee constructor for " << firstName
36       << ' ' << lastName << " called." << endl;
37 } // end Employee constructor
38
39 // destructor deallocates dynamically allocated memory
40 Employee::~Employee()
41 {
42    cout << "~Employee() called for " << firstName
43       << ' ' << lastName << endl;
44
45    delete [] firstName; // release memory
46    delete [] lastName; // release memory
47
48    count--; // decrement static count of employees
49 } // end ~Employee destructor
```

Dynamically allocating char arrays

Non-static member function (i.e., constructor) can modify the class's static data members

Deallocating memory reserved for arrays

# static class member Example

```
50
51 // return first name of employee
52 const char *Employee::getFirstName() const
53 {
54    // const before return type prevents client from modifying
55    // private data; client should copy returned string before
56    // destructor deletes storage to prevent undefined pointer
57    return firstName;
58 } // end function getFirstName
59
60 // return last name of employee
61 const char *Employee::getLastName() const
62 {
63    // const before return type prevents client from modifying
64    // private data; client should copy returned string before
65    // destructor deletes storage to prevent undefined pointer
66    return lastName;
67 } // end function getLastName
```

# static class member Example

```
1  // Fig. 21.23: fig21_23.cpp
2  // Driver to test class Employee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Employee.h" // Employee class definition
8
9  int main()
10 {
11    // use class name and binary scope resolution operator to
12    // access sta
tic number function getCount
13    cout << "Number of employees before instantiation of any objects is "
14       << Employee::getCount() << endl; // use class name
15
16    // use new to dynamically create two new Employees
17    // operator new also calls the object's constructor
18    Employee *e1Ptr = new Employee( "Susan", "Baker" );
19    Employee *e2Ptr = new Employee( "Robert", "Jones" );
20
21    // call getCount on first Employee object
22    cout << "Number of employees after objects are instantiated is "
23       << e1Ptr->getCount();
24
25    cout << "\n\nEmployee 1: "
26       << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
27       << "\nEmployee 2: "
28       << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
```

Calling static member function using class name and binary scope resolution operator

Dynamically creating Employees with new

Calling a static member function through a pointer to an object of the class

# static class member Example

```
29
30    delete e1Ptr; // deallocate memory
31    e1Ptr = 0; // disconnect pointer from free-store space
32    delete e2Ptr; // deallocate memory
33    e2Ptr = 0; // disconnect pointer from free-store space
34
35    // no objects exist, so ca
36    // using the class name and the binary scope resolution operator
37    cout << "Number of employees after objects are deleted is "
38       << Employee::getCount() << endl;
39    return 0;
40 } // end main
```

Releasing memory to which a pointer points

Disconnecting a pointer from any space in memory

```
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 Susan Baker
~Employee() called for Robert Jones
Number of employees after objects are deleted is 0
```

# static Class Members

- Declare a member function 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.
- static data members and static member functions exist independently of any objects of a class.
- When a static member function is called, there might not be any objects of its class in memory!!

# Review of Deeper into C++ Classes

- const objects and const member functions
- Member Composition Example
- friend function Example
- this pointer Example
- Dynamic memory management
  - new and delete operators
- static class members