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.
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.
```cpp
// Fig. 21.1: Time.h
// Definition of class Time.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

class Time
{

public:
    Time(int = 0, int = 0, int = 0); // default constructor

    // set functions
    void setTime(int, int, int); // set time
    void setHour(int); // set hour
    void setMinute(int); // set minute
    void setSecond(int); // set second

    // get functions (normally declared const)
    int getHour() const; // return hour
    int getMinute() const; // return minute
    int getSecond() const; // return second
};
#define TIME_H
```

© 2007 Pearson Ed - All rights reserved.
// print functions (normally declared const)
void printUniversal() const; // print universal time
void printStandard(); // print standard time (should be const)

private:
int hour; // 0 - 23 (24-hour clock format)
int minute; // 0 - 59
int second; // 0 - 59
}; // end class Time

#endif
// Fig. 21.2: Time.cpp
// Member-function definitions for class Time.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

#include "Time.h" // include definition of class Time

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hour, int minute, int second )
{
    setTime( hour, minute, second );
} // end Time constructor

// set hour, minute and second values
void Time::setTime( int hour, int minute, int second )
{
    setHour( hour );
    setMinute( minute );
    setSecond( second );
} // 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
// return minute value
int Time::getMinute() const
{
    return minute;
} // end function getMinute

// return second value
int Time::getSecond() const
{
    return second;
} // end function getSecond

// print Time in universal-time format (HH:MM:SS)
void Time::printUniversal() const
{
    cout << setfill( '0' ) << setw( 2 ) << hour << "":"" << setw( 2 ) << minute << "":"" << setw( 2 ) << second;
} // end function printUniversal

// print Time in standard-time format (HH:MM:SS AM or PM)
void Time::printStandard() // note lack of const declaration
{
    cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << " AM" : " PM" );
} // end function printStandard
// Fig. 21.3: fig21_03.cpp
// Attempting to access a const object with non-const member functions.
#include "Time.h" // include Time class definition

int main()
{
    Time wakeUp(6, 45, 0); // non-constant object
    const Time noon(12, 0, 0); // constant object

    // OBJECT MEMBER FUNCTION
    wakeUp.setHour(18); // non-const non-const
    noon.setHour(12); // const non-const

    wakeUp.getHour(); // non-const const
    noon.getMinute(); // const const
    noon.printUniversal(); // const const
    noon.printStandard(); // const non-const
    return 0;
} // end main
### 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.
// Fig. 21.4: Increment.h
// Definition of class Increment.
#ifndef INCREMENT_H
#define INCREMENT_H

class Increment
{
  public:
    Increment( int c = 0, int i = 1 ); // default constructor

    // function addIncrement definition
    void addIncrement()
    {
      count += increment;
    } // end function addIncrement

    void print() const; // prints count and increment

  private:
    int count;
    const int increment; // const data member
}; // end class Increment

#endif

const data member that must be initialized using a member initializer

© 2007 Pearson Ed - All rights reserved.
// Fig. 21.5: Increment.cpp
// Member-function definitions for class Increment demonstrate using a
// member initializer to initialize a constant of a built-in data type.
#include <iostream>
using std::cout;
using std::endl;

#include "Increment.h" // include definition of class Increment

// constructor
Increment::Increment( int c, int i )
    : count( c ), // initializer for non-const member
      increment( i ), // required initializer for const member
    {
    // empty body
    } // end constructor Increment

// print count and increment values
void Increment::print() const
    {
    cout << "count = " << count << ", increment = " << increment << endl;
    } // end function print
// Fig. 21.6: fig21_06.cpp
// Program to test class Increment.
#include <iostream>
using std::cout;

#include "Increment.h" // include definition of class Increment

int main()
{
    Increment value(10, 5);
    cout << "Before incrementing: ";
    value.print();

    for (int j = 1; j <= 3; j++)
    {
        value.addIncrement();
        cout << "After increment " << j << ": ";
        value.print();
    } // end for

    return 0;
} // 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
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.
Not providing a member initializer for a `const` data member is a compilation error.
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.
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.
A common form of software reusability is **composition**, in which a class has objects of other classes as members.
First example of static data member in C++

```cpp
class Date {
   public:
      static const int monthsPerYear = 12; // number of months in a year
      Date(int m = 1, int d = 1, int y = 1900); // default constructor
      void print() const; // print date in month/day/year format
      ~Date(); // provided to confirm destruction order
   private:
      int month; // 1-12 (January-December)
      int day; // 1-31 based on month
      int year; // any year
      // utility function to check if day is proper for month and year
      int checkDay(int ) const;
}; // end class Date
```

Fig. 10.8 | Date class definition.
```cpp
// Fig. 10.9: Date.cpp
// Date class member-function definitions.
#include <iostream>
#include <stdexcept>
#include "Date.h" // include Date class definition
using namespace std;

// constructor confirms proper value for month; calls
// utility function checkDay to confirm proper value for day
Date::Date( int mn, int dy, int yr )
{
    if ( mn > 0 && mn <= monthsPerYear ) // validate the month
        month = mn;
    else
        throw invalid_argument( "month must be 1-12" );

    year = yr; // could validate yr
    day = checkDay( dy ); // validate the day

    // output Date object to show when its constructor is called
    cout << "Date object constructor for date ";
    print();
    cout << endl;
} // end Date constructor
```

Fig. 10.9  |  Date class member-function definitions. (Part I of 3.)
18.9 Composition Example

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

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

```cpp
// utility function to confirm proper day value based on
// month and year; handles leap years, too
int Date::checkDay( int testDay ) const
{
    static const int daysPerMonth[ monthsPerYear + 1 ] =
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 30, 31 };

    // determine whether testDay is valid for specified month
    if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
        return testDay;

    // February 29 check for leap year
    if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
            ( year % 4 == 0 && year % 100 != 0 ) ) )
        return testDay;

    throw invalid_argument( "Invalid day for current month and year" );
}

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

Copyright © Pearson, Inc. 2013. All Rights Reserved.
18.10 Composition Example

```cpp
// Fig. 10.10: Employee.h
// Employee class definition showing composition.
// Member functions defined in Employee.cpp.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
#include "Date.h" // include Date class definition
using namespace std;
```

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

Fig. 10.10 | Employee class definition showing composition. (Part 2 of 2.)
Member initializers that pass arguments to Date’s implicit default copy constructor

```cpp
#include <iostream>
#include "Employee.h" // Employee class definition
#include "Date.h" // Date class definition
using namespace std;

// constructor uses member initializer list to pass initializer
// values to constructors of member objects
Employee::Employee( const string &first, const string &last,
                     const Date &dateOfBirth, const Date &dateOfHire )
  : firstName( first ), // initialize firstName
    lastName( last ), // initialize lastName
    birthDate( dateOfBirth ), // initialize birthDate
    hireDate( dateOfHire ) // initialize hireDate
{
  // output Employee object to show when constructor is called
  cout << "Employee object constructor: "
       << firstName << ' ' << lastName << endl;
} // end Employee constructor
```

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

```cpp
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** (in 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?

- **Answer:** 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.
Passing objects arguments to a host object constructor.

```cpp
// Fig. 10.12: fig10_12.cpp
// Demonstrating composition—an object with member objects.
#include <iostream>
#include "Employee.h" // Employee class definition
using namespace std;

int main()
{
    Date birth(7, 24, 1949);
    Date hire(3, 12, 1988);
    Employee manager( "Bob", "Blue", birth, hire );

    cout << endl;
    manager.print();
}
```

**Fig. 10.12** Demonstrating composition—an object with member objects. (Part 1 of 2.)
Fig. 10.12 | Demonstrating composition—an object with member objects. (Part 2 of 2.)
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).
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 functions** 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.
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.
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. 21.15: fig21_15.cpp
// Friends can access private members of a class.
#include <iostream>
using std::cout;
using std::endl;

// Count class definition
class Count
{
    friend void setX( Count &, int ); // friend declaration

public:
    // constructor
    Count()
    : x( 0 ) // initialize x to 0
    {
        // empty body
    } // end constructor Count

    // output x
    void print() const
    {
        cout << x << endl;
    } // end function print

private:
    int x; // data member
}; // end class Count

friend function declaration (can appear anywhere in the class)
Fig 18.13 **friend** Function Example

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

- **friend** function can modify Count’s private data
- Calling a **friend** function; note that we pass the **Count** object to the function.
// Fig. 10.16: fig10_16.cpp
// Non-friend/non-member functions cannot access private data of a class.
#include <iostream>
using std::cout;
using std::endl;

// Count class definition (note that there is no friendship declaration)
class Count
{
public:
   // constructor
   Count()
   : x(0) // initialize x to 0
   {
   // empty body
   }
   // end constructor Count

   // output x
   void print() const
   {
      cout << x << endl;
   } // end function print

private:
   int x; // data member
}; // end class Count
Non-friend function cannot access the class’s private data
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
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.
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. 21.17: fig21_17.cpp
// Using the this pointer to refer to object members.
#include <iostream>
using std::cout;
using std::endl;

class Test
{
public:
    Test( int = 0 ); // default constructor
    void print() const;

private:
    int x;
}; // end class Test

// constructor
Test::Test( int value ) : x( value ) // initialize x to value
{
    // empty body
} // end constructor Test
// print x using implicit and explicit this pointers;  
// the parentheses around *this are required  
void Test::print() const  
{
// implicitly use the this pointer to access the member x  
cout << " x = " << x;  

// explicitly use the this pointer and the arrow operator  
// to access the member x  
cout << " this->x = " << this->x;  

// explicitly use the dereferenced this pointer and  
// the dot operator to access the member x  
cout << "(*this).x = " << (*this).x << endl;  
} // end function print

int main()  
{
    Test testObject( 12 ); // instantiate and initialize testObject  
    testObject.print();  
    return 0;  
} // end main

© 2007 Pearson Ed - All rights reserved.

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
Attempting to use the member selection operator (\texttt{.}) with a \texttt{pointer} to an object is a compilation error—the dot member selection operator may be used only with an \texttt{lvalue} such as an \texttt{object's name}, a reference to an \texttt{object} or a dereferenced \texttt{pointer} to an \texttt{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**
  
  ```cpp
  t.setMinute( 30 ).setSecond( 22 );
  ```
  
  - Calls `t.setMinute( 30 );`
  - Then calls `t.setSecond( 22 );`
Cascading Function Calls using the *this* Pointer

```cpp
// Fig. 21.18: Time.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

// Fig. 21.18: Time.h
// Cascading member function calls.

// Time class definition.
// Member functions defined in Time.cpp.
#ifndef TIME_H
#define TIME_H

class Time
{
public:
    Time( int = 0, int = 0, int = 0 ); // default constructor

    // set functions (the Time & return types enable cascading)
    Time &setTime( int, int, int ); // set hour, minute, second
    Time &setHour( int ); // set hour
    Time &setMinute( int ); // set minute
    Time &setSecond( int ); // set second

set functions return Time & (a reference) to enable cascading
```
// get functions (normally declared const)
int getHour() const; // return hour
int getMinute() const; // return minute
int getSecond() const; // return second

// print functions (normally declared const)
void printUniversal() const; // print universal time
void printStandard() const; // print standard time

private:
int hour; // 0 - 23 (24-hour clock format)
int minute; // 0 - 59
int second; // 0 - 59

}; // end class Time

#endif
// Fig. 21.19: Time.cpp
// Member-function definitions for Time class.
#include <iostream>
using std::cout;

#include <iomanip>
using std::setfill;
using std::setw;

#include "Time.h" // Time class definition

// constructor function to initialize private data;
// calls member function setTime to set variables;
// default values are 0 (see class definition)
Time::Time( int hr, int min, int sec )
{
    setTime( hr, min, sec );
} // end Time constructor

// set values of hour, minute, and second
Time &Time::setTime( int h, int m, int s ) // note Time & return
{
    setHour( h );
    setMinute( m );
    setSecond( s );
    return *this; // enables cascading
} // end function setTime

Returning *this pointer enables cascading
Cascading Function Calls using the this Pointer

// set hour value
Time &Time::setHour(int h) // note Time & return
{
    hour = (h >= 0 && h < 24) ? h : 0; // validate hour
    return *this; // enables cascading
} // end function setHour

// set minute value
Time &Time::setMinute(int m) // note Time & return
{
    minute = (m >= 0 && m < 60) ? m : 0; // validate minute
    return *this; // enables cascading
} // end function setMinute

// set second value
Time &Time::setSecond(int s) // note Time & return
{
    second = (s >= 0 && s < 60) ? s : 0; // validate second
    return *this; // enables cascading
} // end function setSecond

// get hour value
int Time::getHour() const
{
    return hour;
} // end function getHour
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

```cpp
// Fig. 21.20: fig21_20.cpp
// Cascading member function calls with the this pointer.
#include <iostream>
using std::cout;
using std::endl;
#include "Time.h" // Time class definition

int main()
{
    Time t; // create Time object

    // cascaded function calls
    t.setHour(18).setMinute(30).setSecond(22);

    // output time in universal and standard formats
    cout << "Universal time: ";
    t.printUniversal();

    cout << "\nStandard time: ";
    t.printStandard();

    cout << "\n\nNew standard time: ";
    // cascaded function calls
    t.setTime(20, 20, 20).printStandard();
    cout << endl;
    return 0;
} // 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 \textit{this} Pointer

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

New standard time: 8:20:20 PM
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.
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:**

```cpp
Time *timePtr
timePtr = new Time;
```
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:**

```cpp
delete timePtr;
```
19.9 Operators new and delete

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

Example

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

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

Example

```cpp
Time *timePtr = new Time( 12, 45, 0 );
```
- **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[stores];
    ```
Delete a dynamically allocated array:

```cpp
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.
The following example is older and from the 5\textsuperscript{th} 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 7\textsuperscript{th} Edition of Deitel & Deitel.
// Fig. 21.21: Employee.h
// Employee class definition.
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

class Employee
{
public:
  Employee(const char * const, const char * const); // constructor
  ~Employee(); // destructor
  const char *getFirstName() const; // return first name
  const char *getLastName() const; // return last name

  // static member function
  static int getCount(); // return number of objects instantiated

private:
  char *firstName;
  char *lastName;

  // static data
  static int count; // number of objects instantiated
}; // end class Employee

#endif
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.
// constructor dynamically allocates space for first and last name and
// uses strcpy to copy first and last names into the object
Employee::Employee( const char * const first, const char * const last )
{
    firstName = new char[ strlen( first ) + 1 ];
    strcpy( firstName, first );

    lastName = new char[ strlen( last ) + 1 ];
    strcpy( lastName, last );

    count++; // increment static count of employees

    cout << "Employee constructor for " << firstName
     << ' ' << lastName << " called." << endl;
} // end Employee constructor

// destructor deallocates dynamically allocated memory
Employee::~Employee()
{
    cout << "~Employee() called for " << firstName
     << ' ' << lastName << endl;

    delete [] firstName; // release memory
    delete [] lastName; // release memory

    count--; // decrement static count of employees
} // end ~Employee destructor
// return first name of employee
const char *Employee::getFirstName() const {
    // const before return type prevents client from modifying
    // private data; client should copy returned string before
    // destructor deletes storage to prevent undefined pointer
    return firstName;
} // end function getFirstName

// return last name of employee
const char *Employee::getLastName() const {
    // const before return type prevents client from modifying
    // private data; client should copy returned string before
    // destructor deletes storage to prevent undefined pointer
    return lastName;
} // end function getLastName
```cpp
// Fig. 21.23: fig21_23.cpp
// Driver to test class Employee.
#include <iostream>
using std::cout;
using std::endl;

#include "Employee.h" // Employee class definition

int main()
{
    // use class name and binary scope resolution operator to
    // access static number function getCount
    cout << "Number of employees before instantiation of any objects is "
         << Employee::getCount() << endl; // use class name

    // use new to dynamically create two new Employees
    // operator new also calls the object's constructor
    Employee *e1Ptr = new Employee("Susan", "Baker");
    Employee *e2Ptr = new Employee("Robert", "Jones");

    // call getCount on first Employee object
    cout << "Number of employees after objects are instantiated is "
         << e1Ptr->getCount();

    cout << "\n\nEmployee 1: "
         << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
         << "\n\nEmployee 2: "
         << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
```
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

Releasing memory to which a pointer points
Disconnecting a pointer from any space in memory
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