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
- Abstract Data Types
21.2 \texttt{const} (Constant) Objects and \texttt{const} Member Functions

- Principle of least privilege
  - One of the most fundamental principles of good software engineering
  - Applies to objects, too

- \texttt{const} objects
  - Keyword \texttt{const}
  - Specifies that an object is not modifiable.
  - Attempts to modify the object will result in compilation errors.

Example

- \texttt{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 21.2

- 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

```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

#endif // TIME_H
```
const Example

// 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

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

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

// 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 << ":" << setfill( '0' ) << setw( 2 ) << minute << ":" << setfill( '0' ) << 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 ) << ":" << setfill( '0' ) << setw( 2 ) << minute
    << ":" << setfill( '0' ) << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
} // end function printStandard
const Example

```cpp
// 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
```

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.
// 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
// 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
}

// print count and increment values
void Increment::print() const
{
  cout << "count = " << count <<", increment = " << increment << endl;
}

Colon (:) marks the start of a member initializer list
Member initializer for non-const member count
Required member initializer for const member increment

© 2007 Pearson Ed - All rights reserved.
// Fig. 21.6: fig21_06.cpp
// Program to test class Increment.
#include <iostream>
#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.
Common Programming Error 21.5

- Not providing a member initializer for a `const` data member is a compilation error.
Software Engineering Observation 21.4

- Constant data members (\texttt{const} objects and \texttt{const} variables) and data members declared as \texttt{references} must be initialized with member initializer syntax; assignments for these types of data in the constructor body are not allowed.
21.3 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 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 21.5

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

```cpp
// Fig. 21.10: Date.h
// Date class definition; Member functions defined in Date.cpp
#ifndef DATE_H
#define DATE_H

class Date
{
public:
    Date( int = 1, int = 1, int = 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

#endif
```

© 2007 Pearson Ed - All rights reserved.
Composition Example

```cpp
1 // Fig. 21.11: Date.cpp
2 // Member-function definitions for class Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // include Date class definition
8
9 // constructor confirms proper value for month; calls
10 // utility function checkDay to confirm proper value for day
11 Date::Date( int mn, int dy, int yr )
12 {
13     if ( mn > 0 && mn <= 12 ) // validate the month
14         month = mn;
15     else
16         { // invalid month set to 1
17             month = 1; // invalid month set to 1
18             cout << "Invalid month (" << mn << ") set to 1.\n";
19         } // end else
20     year = yr; // could validate yr
21     day = checkDay( dy ); // validate the day
22
23     // output Date object to show when its constructor is called
24     cout << "Date object constructor for date ";
25     print();
26     cout << endl;
27 } // end Date constructor
```
Composition Example

29
30 // print Date object in form month/day/year
31 void Date::print() const
32 {
33   cout << month << '/' << day << '/' << year;
34 } // end function print
35
36 // output Date object to show when its destructor is called
37 Date::~Date() {
38   {
39     cout << "Date object destructor for date ";
40     print();
41     cout << endl;
42 } // end ~Date destructor
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[13] =
    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 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;

    cout << "Invalid day (" << testDay << ") set to 1.\n";
    return 1; // leave object in consistent state if bad value
}
```
Composition Example

1 // Fig. 21.12: Employee.h
2 // Employee class definition.
3 // Member functions defined in Employee.cpp.
4 ifndef EMPLOYEE_H
5 define EMPLOYEE_H
6
7 #include "Date.h" // include Date class definition
8
9 class Employee
10 {
11 public:
12 Employee( const char * const, const char * const,
13 const Date &, const Date &);
14 void print() const;
15 ~Employee(); // provided to confirm destruction
16 private:
17 char firstName[ 25 ];
18 char lastName[ 25 ];
19 const Date birthDate; // composition: member object
20 const Date hireDate; // composition: member object
21 }; // end class Employee
22
23 endif

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

const objects of class Date as members
Composition Example

```
1 // Fig. 21.13: 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 strncpy prototypes
8 using std::strlen;
9 using std::strncpy;
10
11 #include "Employee.h" // Employee class definition
12 #include "Date.h" // Date class definition
13
14 // constructor uses member initializer list to pass initializer
15 // values to constructors of member objects birthDate and hireDate
16 // [Note: This invokes the so-called "default copy constructor" which the
17 // C++ compiler provides implicitly.]
18 Employee::Employee( const char * const first, const char * const last,
19                 const Date &dateOfBirth, const Date &dateOfHire )
20     : birthDate( dateOfBirth ), // initialize birthDate
21     hireDate( dateOfHire )     // initialize hireDate
22 {
23     // copy first into firstName and be sure that it fits
24     int length = strlen( first );
25     length = ( length < 25 ? length : 24 );
26     strncpy( firstName, first, length );
27     firstName[ length ] = '\0';
```
Composition Example

28  // copy last into lastName and be sure that it fits
29  length = strlen(last);
30  length = (length < 25 ? length : 24);
31  strncpy(lastName, last, length);
32  lastName[length] = '\0';
33
34  // output Employee object to show when constructor is called
35  cout << "Employee object constructor: "
36     << firstName << ' ' << lastName << endl;
37 } // end Employee constructor
38
39  // print Employee object
40  void Employee::print() const
41  {
42     cout << lastName <<", " << firstName <<" Hired: ";
43     hireDate.print();
44     cout << " Birthday: ";
45     birthDate.print();
46     cout << endl;
47 } // end function print
48
49  // output Employee object to show when its destructor is called
50  Employee::~Employee()
51  {
52     cout << "Employee object destructor: "
53     << lastName <<", " << firstName << endl;
54 } // end ~Employee destructor
Composition Example

```cpp
// Fig. 21.14: fig21_14.cpp
// Demonstrating composition--an object with member objects.
#include <iostream>
using std::cout;
using std::endl;

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

int main()
{
    Date birth( 7, 24, 1949 );
    Date hire( 3, 12, 1988 );
    Employee manager( "Bob", "Blue", birth, hire );
    cout << endl;
    manager.print();
    cout << endl;
    cout << "Test Date constructor with invalid values:\n";
    Date lastDayOff( 14, 35, 1994 ); // invalid month and day
    cout << endl;
    return 0;
} // end main
```

Passing objects to a host object constructor
Composition Example

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


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
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).
21.4 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 or entire classes may be declared to be friends of a class.
  - 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)
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     cout << "counter.x after instantiation: ";
39     counter.print();
40     setX( counter, 8 ); // set x using a friend function
41     cout << "counter.x after call to setX friend function: ";
42     counter.print();
43     return 0;
44 } // 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
friend Function Example

// 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
{
    // 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
// function cannotSetX tries to modify private data of Count,
// but cannot because the function is not a friend of Count
void cannotSetX( Count &c, int val )
{
    c.x = val; // ERROR: cannot access private member in Count
} // end function cannotSetX

int main()
{
    Count counter; // create Count object
    cannotSetX( counter, 3 ); // cannotSetX is not a friend
    return 0;
} // end main

Non-\texttt{friend} function cannot access the class’s \texttt{private} data
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
```
21.5 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.
21.5 Using the this Pointer

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

© 2007 Pearson Ed - All rights reserved.
// 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
Example

```cpp
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 << "    \n this->x = " << this->x;

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

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

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
Common Programming Error 21.7

- 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 the dereferenced `this` pointer
- Example
  - `t.setMinute(30).setSecond(22);`
    - Calls `t.setMinute(30);`
    - Then calls `t.setSecond(22);`
// 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 & to enable cascading
Cascading Function Calls using this Pointer

19  // get functions (normally declared const)
20  int getHour() const; // return hour
21  int getMinute() const; // return minute
22  int getSecond() const; // return second
23
24  // print functions (normally declared const)
25  void printUniversal() const; // print universal time
26  void printStandard() const; // print standard time
27
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 this Pointer

// 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
data
} // end function setTime

Returning dereferenced this pointer enables cascading
Cascading Function Calls using 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
Systems Programming: Deeper into C++ Classes

Cascading Function Calls using this Pointer

```cpp
55 // get minute value
56 int Time::getMinute() const
57 {
58     return minute;
59 } // end function getMinute
60
61 // get second value
62 int Time::getSecond() const
63 {
64     return second;
65 } // end function getSecond
66
67 // print Time in universal-time format (HH:MM:SS)
68 void Time::printUniversal() const
69 {
70     cout << setfill( '0' ) << setw( 2 ) << hour << " :
71         << setw( 2 ) << minute << " :
72         << 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 this Pointer

// 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 << endl;
    cout << "Standard time: ";
    t.printStandard();
    cout << endl;
    cout << "New 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 this Pointer

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

New standard time: 8:20:20 PM
21.6 Dynamic Memory Management: Operators \texttt{new} and \texttt{delete}

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

- **Operator `new`**
  - Allocates (i.e., reserves) storage of the proper size for an object at execution time
  - Calls a constructor to initialize the object.
  - Returns a pointer of the type specified to the right of `new`.
  - 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)
  - Region of memory assigned to each program for storing objects created at execution time.

**Example:**

```
Time *timePtr
timePtr = new Time;
```
Operators \texttt{new} and \texttt{delete}

- **Operator \texttt{delete}**
  - Destroys a dynamically allocated object.
  - Calls the destructor for the object.
  - Deallocates (i.e., releases) memory from the free store.
  - The memory can then be reused by the system to allocate other objects.

\textbf{Example:}

\texttt{delete timePtr;}
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 );`
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.
    ```
    mulePtr *Mule = new Mules[mules_in];
    ```
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
  * Only the first object in the array would have a destructor call.
21.7 static Class Members

- **static** data member
  - Only one copy of a variable 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.
  - Declaration begins with keyword `static`
**static Class Members**

- **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 has class scope.
  - Can be declared `public`, `private` or `protected`.

© 2007 Pearson Ed - All rights reserved.
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).

- const static 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.
**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`
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.
static class member Example

```cpp
// 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
```

Function prototype for static member function

**static data member** keeps track of number of Employee objects that currently exist.
// Fig. 21.22: Employee.cpp
// Member-function definitions for class Employee.
#include <iostream>
using std::cout;
using std::endl;

#include <cstring> // strlen and strcpy prototypes
using std::strlen;
using std::strcpy;

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

// define and initialize static data member at file scope
int Employee::count = 0;

// define static member function that returns number of
// Employee objects instantiated (declared static in Employee.h)
int Employee::getCount()
{
    return count;
} // end static function getCount
// 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
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
# Fig. 21.23: fig21_23.cpp
// Driver to test class Employee.

```cpp
#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
  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 << "
Employee 1: " << e1Ptr->getFirstName() << " " << e1Ptr->getLastName() << "
Employee 2: " << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n";
```

© 2007 Pearson Ed - All rights reserved.
static class member Example

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.
Abstract data types (ADTs)

- Essentially ways of representing real-world notions to some satisfactory level of precision within a computer system.

- Types like `int`, `double`, `char` and others are all ADTs.
  - e.g., `int` is an abstract representation of an integer.

- Captures two notions:
  - Data representation
  - Operations that can be performed on the data.

- **C++ classes implement ADTs and their services.**
Array Abstract Data Type

- Many array operations not built into C++
  - e.g., subscript range checking
- Programmers can develop an array ADT as a class that is preferable to “raw” arrays
  - Can provide many helpful new capabilities
- C++ Standard Library class template `vector`.
Summary

- `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
- Abstract Data Types