Classes Part II

`const` (Constant) Objects and `const` Member Functions

- Principle of least privilege

  Only allow modification of necessary objects

- Keyword **`const`**
  - Specify object not modifiable
  - Compiler error if attempt to modify **`const`** object
  - Example:

```
const Time noon( 12, 0, 0 );
```

Declares **`const`** object **`noon`** of class **`Time`**

Initializes to 12

`const` (Constant) Objects and `const` Member Functions

- **`const`** member functions
  - Member functions for **`const`** objects must also be **`const`**
    - Cannot modify object
- Specify **`const`** in both prototype and definition

  Prototype

  After parameter list

  Definition

  Before beginning left brace
- Constructors and destructors

  Cannot be **`const`**:

  Must be able to modify objects

  Constructor

  Initializes objects

  Destructor

  Performs termination housekeeping

```
1   // Fig. 7.1: time5.h
2   // Definition of class Time.
3   // Member functions defined in time5.cpp.
4   #ifndef TIME5_H
5   #define TIME5_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 min
16    void setSecond( int );          // set sec
17
18    // get functions (normally declared const)
19    int getHour() const;            // return hour
20    int getMinute() const;          // return mi
21    int getSecond() const;          // return se
22
23    // print functions (normally declared const)
24    void printUniversal() const;    // print universal time
25    void printStandard();           // print standard time
26
27  private:
28    int hour;     // 0 - 23 (24-hour clock format)
29    int minute;   // 0 - 59
30    int second;   // 0 - 59
31
32  }; // end class Time
33
34  #endif
```

Declare **const** get functions

Declare **const** function **printUniversal**

const (Constant) Objects and const Member Functions

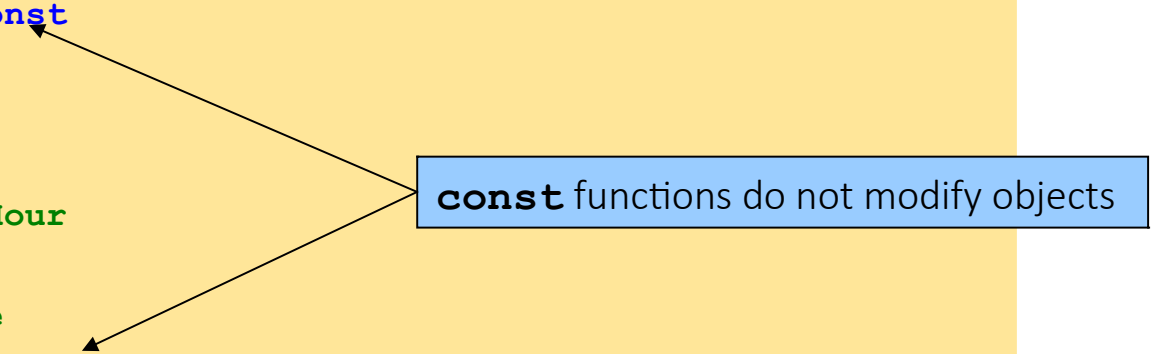
```
1     // Fig. 7.2: time5.cpp
2     // Member-function definitions for class Time.
3     #include <iostream>
4
5     using std::cout;
6
7     #include <iomanip>
8
9     using std::setfill;
10    using std::setw;
11
12    // include definition of class Time from time5.h
13    #include "time5.h"
14
15    // constructor function to initialize private data;
16    // calls member function setTime to set variables;
17    // default values are 0 (see class definition)
18    Time::Time( int hour, int minute, int second )
19    {
20       setTime( hour, minute, second );
21
22    } // end Time constructor
23
```

const (Constant) Objects and const Member Functions

```
24    // set hour, minute and second values
25    void Time::setTime( int hour, int minute, int second )
26    {
27       setHour( hour );
28       setMinute( minute );
29       setSecond( second );
30
31    } // end function setTime
32
33    // set hour value
34    void Time::setHour( int h )
35    {
36       hour = ( h >= 0 && h < 24 ) ? h : 0;
37
38    } // end function setHour
39
40    // set minute value
41    void Time::setMinute( int m )
42    {
43       minute = ( m >= 0 && m < 60 ) ? m : 0;
44
45    } // end function setMinute
46
```

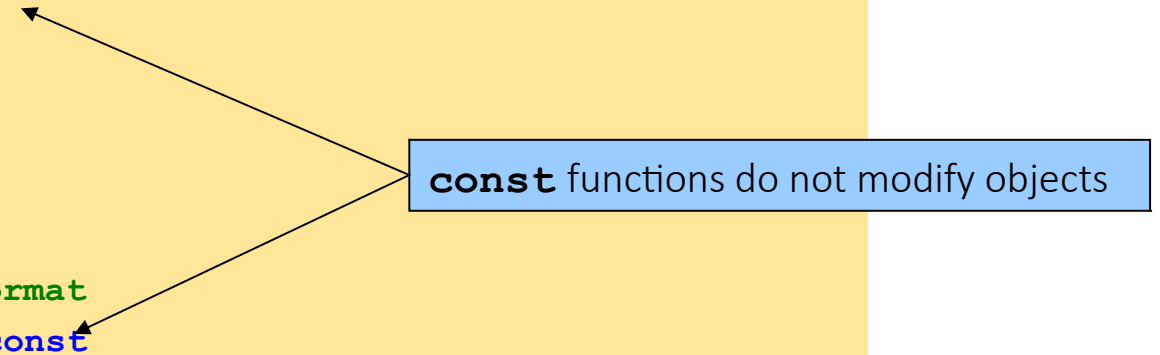
`const` (Constant) Objects and `const` Member Functions

```
47    // set second value
48    void Time::setSecond( int s )
49    {
50       second = ( s >= 0 && s < 60 ) ? s : 0;
51
52    } // end function setSecond
53
54    // return hour value
55    int Time::getHour() const
56    {
57       return hour;
58
59    } // end function getHour
60
61    // return minute value
62    int Time::getMinute() const
63    {
64       return minute;
65
66    } // end function getMinute
67
```



`const` (Constant) Objects and `const` Member Functions

```
68    // return second value
69    int Time::getSecond() const
70    {
71       return second;
72
73    } // end function getSecond
74
75    // print Time in universal format
76    void Time::printUniversal() const
77    {
78       cout << setfill( '0' ) << setw( 2 ) << hour << ":"
79            << setw( 2 ) << minute << ":"
80            << setw( 2 ) << second;
81
82    } // end function printUniversal
83
84    // print Time in standard format
85    void Time::printStandard() // note lack of const declaration
86    {
87       cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
88            << ":" << setfill( '0' ) << setw( 2 ) << minute
89            << ":" << setw( 2 ) << second
90            << ( hour < 12 ? " AM" : " PM" );
91
92    } // end function printStandard
```



`const` (Constant) Objects and `const` Member Functions

```
1       // Fig. 7.3: fig07_03.cpp
2       // Attempting to access a const object with
3       // non-const member functions.
4
5       // include Time class definition from time5.h
6       #include "time5.h"
7
8       int main()
9       {
10        Time wakeUp( 6, 45, 0 );        // non-constant object
11        const Time noon( 12, 0, 0 );    // constant object
12
```

Declare **noon** as a **const** object

Note that non-const constructor can initialize **const** object

const (Constant) Objects and const Member Functions

```
13                                  // OBJECT      MEMBER FUNCTION
14        wakeUp.setHour( 18 );     // non-const   non-const
15
16        noon.setHour( 12 );       // const       non-const
17
18        wakeUp.getHour();         // non-const   const
19
20        noon.getMinute();         // const       c
21        noon.printUniversal();    // const       const
22
23        noon.printStandard();     // const       non-const
24
25        return 0;
26
27     } // end main
```

Attempting to invoke non-**const** member function on **const** object results in compiler error

Attempting to invoke non-**const** member function on **const** object results in compiler error even if function does not modify object

```
d:\cpphtp4_examples\ch07\fig07_01\fig07_01.cpp(16) : error C2662:
   'setHour' : cannot convert 'this' pointer from 'const class Time'
   to 'class Time &'
        Conversion loses qualifiers
d:\cpphtp4_examples\ch07\fig07_01\fig07_01.cpp(23) : error C2662:
   'printStandard' : cannot convert 'this' pointer from 'const class
   Time' to 'class Time &'
        Conversion loses qualifiers
```

`const` (Constant) Objects and `const` Member Functions

- Member initializer syntax
  - Initializing with member initializer syntax
    - Can be used for

      all data members
    - Must be used for

      `const` data members

      Data members that are references

const (Constant) Objects and const Member Functions

```
1     // Fig. 7.4: fig07_04.cpp
2     // Using a member initializer to initialize a
3     // constant of a built-in data type.
4     #include <iostream>
5
6     using std::cout;
7     using std::endl;
8
9     class Increment {
10
11    public:
12       Increment( int c = 0, int i = 1 );  // default constructor
13
14       void addIncrement()
15       {
16          count += increment;
17       } // end function addIncrement
18
19      void print() const;     // prints count and increment
20
21
```

const (Constant) Objects and const Member Functions

```
22    private:
23       int count;
24       const int increment;    // const data member
25
26    }; // end class Increment
27
28    // constructor
29    Increment::Increment( int c, int i )
30       : count( c ),     // initializer for non-const member
31         increment( i )  // required initializer for const member
32    {
33       // empty body
34
35    } // end Increment constructor
36
37    // print count and increment values
38    void Increment::print() const
39    {
40       cout << "count = " << count
41            << ", increment = " << increment << endl;
42
43    } // end function print
44
```

const (Constant) Objects and const Member Functions

```
22    private:
23       int count;
24       const int increment;    // const data member
25
26    }; // end class Increment
27
28    // constructor
29    Increment::Increment( in
30       : count( c ),      // initializer for non-const member
31         increment( i )   // required initializer for const member
32    {
33       // empty body
34
35    } // end Increment constructor
36
37    // print count and increment
38    void Increment::print() const
39    {
40       cout << "count = " << count
41            << ", increment = " << increment << endl;
42
43    } // end function print
44
```

Declare increment as **const** data member

Member initializer list separated from parameter list by colon

Member initializer syntax can be used for non-**const** data member **count**

Member initializer syntax must be used for **const** data member **increment**

Member initializer consists of data member name (**increment**) followed by parentheses containing initial value (**i**).

const (Constant) Objects and const Member Functions

```
28    // constructor
29    Increment::Increment( int c, int i )
30       : count( c ),     // initializer for non-const member
31         increment( i )  // required initializer for const member
32    {
33       // empty body
```

```
45    int main()
46    {
47       Increment value( 10, 5 );
48
49       cout << "Before incrementing: ";
50       value.print();
51
52       for ( int j = 0; j < 3; j++ ) {
53          value.addIncrement();
54          cout << "After increment " << j + 1 << ": ";
55          value.print();
56       }
57
58       return 0;
59
60    } // 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
```

const (Constant) Objects and const Member Functions

```
1       // Fig. 7.5: fig07_05.cpp
2       // Attempting to initialize a constant of
3       // a built-in data type with an assignment.
4       #include <iostream>
5
6       using std::cout;
7       using std::endl;
8
9       class Increment {
10
11    public:
12       Increment( int c = 0, int i = 1 );  // default constructor
13
14       void addIncrement()
15       {
16          count += increment;
17
18       } // end function addIncrement
19
20       void print() const;     // prints count and increment
21
```

const (Constant) Objects and const Member Functions

```
22    private:
23       int count;
24       const int increment;     // const data member
25
26    }; // end class Increment
27
28    // constructor
29    Increment::Increment( int c, int i )
30    {              // Constant member 'increme
31       count = c;       // allowed because cou
32       increment = i;   // ERROR: Cannot modify a const object
33
34    } // end Increment constructor
35
36    // print count and increment values
37    void Increment::print() const
38    {
39       cout << "count = " << count
40            << ", increment = " << increment << endl;
41
42    } // end function print
43
```

Declare increment as **const** data member

Attempting to modify **const** data member **increment** results in error.

`const` (Constant) Objects and `const` Member Functions

```
44    int main()
45    {
46       Increment value( 10, 5 );
47
48       cout << "Before incrementing: ";
49       value.print();
50
51       for ( int j = 0; j < 3; j++ ) {
52          value.addIncrement();
53          cout << "After increment " << j + 1 << ": ";
54          value.print();
55       }
56
57       return 0;
58
59    } // end main
```

```
D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(30) : error C2758:
   'increment' : must be initialized in constructor base/member
   initializer list
        D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(24) :
           see declaration of 'increment'
D:\cpphtp4_examples\ch07\Fig07_03\Fig07_03.cpp(32) : error C2166:
   l-value specifies const object
```

Not using member initializer syntax to initialize **`const`** data member **`increment`** results in error.

Attempting to modify **`const`** data member **`increment`** results in error.

Composition: Objects as Members of Classes

- Composition
  - Class has objects of other classes as members
- Construction of objects
  - Member objects constructed in order declared
    - Not in order of constructor's member initializer list
    - Constructed before enclosing class objects (host objects)

Composition: Objects as Members of Classes

```
1     // Fig. 7.6: date1.h
2     // Date class definition.
3     // Member functions defined in date1.cpp
4     #ifndef DATE1_H
5     #define DATE1_H
6
7     class Date {
8
9     public:
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
14    private:
15      int month;  // 1-12 (January-December)
16      int day;    // 1-31 based on month
17      int year;   // any year
18
19      // utility function to test proper day for month and year
20      int checkDay( int ) const;
21
22    }; // end class Date
23
24    #endif
```

Note no constructor with parameter of type **`Date`**.
Recall compiler provides default copy constructor.

Composition: Objects as Members of Classes

```
1     // Fig. 7.7: date1.cpp
2     // Member-function definitions for class Date.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     // include Date class definition from date1.h
9     #include "date1.h"
10
11    // constructor confirms proper value for month; calls
12    // utility function checkDay to confirm proper value for day
13    Date::Date( int mn, int dy, int yr )
14    {
15       if ( mn > 0 && mn <= 12 )  // validate the month
16          month = mn;
17
18       else {                     // invalid month set to 1
19          month = 1;
20          cout << "Month " << mn << " invalid. Set to month 1.\n";
21       }
22
23       year = yr;                 // should validate yr
24       day = checkDay( dy );      // validate the day
25
```

Composition: Objects as Members of Classes

```
26       // output Date object to show when its constructor is called
27       cout << "Date object constructor for date ";
28       print();
29       cout << endl;
30
31    } // end Date constructor
32
33    // print Date object in form month/day/year
34    void Date::print() const
35    {
36       cout << month << '/' << day << '/' << year;
37
38    } // end function print
39
40    // output Date object to show when its destructor is called
41    Date::~Date()
42    {
43       cout << "Date object destructor for date ";
44       print();
45       cout << endl;
46
47    } // end destructor ~Date
48
```

Output to show timing of constructors

No arguments; each member function contains implicit handle to object on which it operates.

Output to show timing of destructors.

Composition: Objects as Members of Classes

```
49    // utility function to confirm proper day value based on
50    // month and year; handles leap years, too
51    int Date::checkDay( int testDay ) const
52    {
53       static const int daysPerMonth[ 13 ] =
54          { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
55
56       // determine whether testDay is valid for specified month
57       if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
58          return testDay;
59
60       // February 29 check for leap year
61       if ( month == 2 && testDay == 29 &&
62          ( year % 400 == 0 ||
63             ( year % 4 == 0 && year % 100 != 0 ) ) )
64          return testDay;
65
66       cout << "Day " << testDay << " invalid. Set to day 1.\n";
67
68       return 1;  // leave object in consistent state if bad value
69
70    } // end function checkDay
```

Composition: Objects as Members of Classes

```
1     // Fig. 7.8: employee1.h
2     // Employee class definition.
3     // Member functions defined in employee1.cpp.
4     #ifndef EMPLOYEE1_H
5     #define EMPLOYEE1_H
6
7     // include Date class definition from date1.h
8     #include "date1.h"
9
10    class Employee {
11
12    public:
13       Employee(
14          const char *, const char *, const Date &, const Date & );
15
16       void print() const;
17       ~Employee();  // provided to confirm destruction order
18
19    private:
20       char firstName[ 25 ];
21       char lastName[ 25 ];
22       const Date birthDate;  // composition: member object
23       const Date hireDate;   // composition: member object
24
25    }; // end class Employee
26
27    #endif
```

Using composition; **Employee** object contains **Date** objects as data members

```
1   // Fig. 7.9: employee1.cpp
2   // Member-function definitions for class Employee.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <cstring>      // strcpy and strlen prototypes
9
10  #include "employee1.h"  // Employee class definition
11  #include "date1.h"      // Date class definition
12
13   // constructor uses member initializer list to pass initializer
14   // values to constructors of member objects birthDate and
15   // hireDate [Note: This invokes the so-called "default copy
16   // constructor" which the C++ compiler provides implicitly.]
17   Employee::Employee( const char *first, const char *last,
18      const Date &dateOfBirth, const Date &dateOfHire )
19      : birthDate( dateOfBirth ),   // initialize birthDate
20        hireDate( dateOfHire )      // initialize hireDate
21   {
22      // copy first into firstName and be sure th
23      int length = strlen( first );
24      length = ( length < 25 ? length : 24 );
25      strncpy( firstName, first, length );
26      firstName[ length ] = '\0';
27
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;
```

Member initializer syntax to initialize **`Date`** data members **`birthDate`** and **`hireDate`**; compiler uses default copy constructor.

Output to show timing of constructors.

Composition: Objects as Members of Classes

```
38    } // end Employee constructor
39
40    // print Employee object
41    void Employee::print() const
42    {
43       cout << lastName << ", " << firstName << "\nHired: ";
44       hireDate.print();
45       cout << "  Birth date: ";
46       birthDate.print();
47       cout << endl;
48
49    } // end function print
50
51    // output Employee object to show when
52    Employee::~Employee()
53    {
54       cout << "Employee object destructor: "
55            << lastName << ", " << firstName << endl;
56
57    } // end destructor ~Employee
```

Output to show timing of destructors.

Composition: Objects as Members of Classes

```
1   // Fig. 7.10: fig07_10.cpp
2   // Demonstrating composition--an object with member objects.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include "employee1.h"  // Employee class definition
9
10  int main()
11  {
12     Date birth( 7, 24, 1949 );
13     Date hire( 3, 12, 1988 );
14     Employee manager( "Bob", "Jones", birth, hire );
15
16     cout << '\n';
17     manager.print();
18
19     cout << "\nTest Date constructor with invalid values:\n";
20     Date lastDayOff( 14, 35, 1994 );  // invalid month and day
21     cout << endl;
22
23     return 0;
24
25  } // end main
```

Create **Date** objects to pass to **Employee** constructor.

Composition: Objects as Members of Classes

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

Jones, Bob
Hired: 3/12/1988  Birth date: 7/24/1949

Test Date constructor with invalid values:
Month 14 invalid. Set to month 1.
Day 35 invalid. Set to day 1.
Date object constructor for date 1/1/1994

Date object destructor for date 1/1/1994
Employee object destructor: Jones, 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
```

Note two additional **Date** objects constructed; no output since default copy constructor used.

Destructor for host object **manager** runs before destructors for member objects **hireDate** and **birthDate**.

Destructor for **Employee**'s member object **hireDate**.

Destructor for **Employee**'s member object **birthDate**.

Destructor for **Date** object **hire**.

Destructor for **Date** object **birth**.

`friend` Functions and `friend` Classes

- **`friend`** function
  - Defined outside class's scope
  - Right to access non-public members
- Declaring **`friend`**s
  - Function
    - Precede function prototype with keyword **`friend`**
  - All member functions of class **`ClassTwo`** as **`friend`**s of class **`ClassOne`**
    - Place declaration of form

      ```
      friend class ClassTwo;
      ```

      in **`ClassOne`** definition

`friend` Functions and `friend` Classes

- Properties of friendship
  - Friendship granted, not taken
    - Class **`B` `friend`** of class **`A`**
      - Class **`A`** must explicitly declare class **`B` `friend`**
    - Not symmetric
      - Class **`B` `friend`** of class **`A`**
      - Class **`A`** not necessarily **`friend`** of class **`B`**
    - Not transitive
      - Class **`A` `friend`** of class **`B`**
      - Class **`B` `friend`** of class **`C`**
      - Class **`A`** not necessarily **`friend`** of Class **`C`**

`friend` Functions and `friend` Classes

```
1       // Fig. 7.11: fig07_11.cpp
2       // Friends can access private members of a class.
3       #include <iostream>
4
5       using std::cout;
6       using std::endl;
7
8       // Count class definition
9       class Count {
10        friend void setX( Count &, int ); // friend declaration
11
12    public:
13
14       // constructor
15       Count()
16          : x( 0 )  // initialize x to 0
17       {
18          // empty body
19
20       } // end Count constructor
21
```

Precede function prototype with keyword **`friend`**.

`friend` Functions and `friend` Classes

```
22       // output x
23       void print() const
24       {
25          cout << x << endl;
26
27       } // end function print
28
29    private:
30       int x;  // data member
31
32    }; // end class Count
33
34    // function setX can modify private data of Count
35    // because setX is declared as a
36    void setX( Count &c, int val )
37    {
38       c.x = val;  // legal: setX is a friend of Count
39
40    } // end function setX
41
```

Pass **Count** object since C-style, standalone function

Since **setX friend** of **Count**, can access and modify **private** data member **x**.

`friend` Functions and `friend` Classes

```
42    int main()
43    {
44       Count counter;        // create Count object
45
46       cout << "counter.x after instantiat
47       counter.print();
48
49       setX( counter, 8 );  // set x with a friend
50
51       cout << "counter.x after call to setX friend function: ";
52       counter.print();
53
54       return 0;
55
56    } // end main
```

Use **`friend`** function to access and modify **`private`** data member **`x`**.

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

`friend` Functions and `friend` Classes

```
1     // Fig. 7.12: fig07_12.cpp
2     // Non-friend/non-member functions cannot access
3     // private data of a class.
4     #include <iostream>
5
6     using std::cout;
7     using std::endl;
8
9     // Count class definition
10    // (note that there is no friendship declaration)
11    class Count {
12
13    public:
14
15       // constructor
16       Count()
17          : x( 0 )  // initialize x to 0
18       {
19          // empty body
20
21       } // end Count constructor
22
```

`friend` Functions and `friend` Classes

```
23       // output x
24       void print() const
25       {
26          cout << x << endl;
27
28       } // end function print
29
30    private:
31       int x;  // data member
32
33    }; // end class Count
34
35    // function tries to modify pr
36    // but cannot because function
37    void cannotSetX( Count &c, int val )
38    {
39       c.x = val;  // ERROR: cannot access private member in Count
40
41    } // end function cannotSetX
42
43    int main()
44    {
45       Count counter;             // create Count object
46
47       cannotSetX( counter, 3 ); // cannotSetX is not a friend
48
49       return 0;
50
51    } // end main
```

Attempting to modify **`private`** data member from non-**`friend`** function results in error.

`friend` Functions and `friend` Classes

```
23       // output x
24       void print() const
25       {
26          cout << x << endl;
27
28       } // end function print
29
30    private:
31       int x;  // data member
32
33    }; // end class Count
34
35    // function tries to modify private data of Count,
36    // but cannot because function is not a friend of Count
37    void cannotSetX( Count &c, int val )
38    {
39       c.x = val;  // ERROR: cannot access private member in Count
40
41    } // end function cannotSetX
```

```
D:\cpphtp4_examples\ch07\Fig07_12\Fig07_12.cpp(39) : error C2248:
   'x' : cannot access private member declared in class 'Count'
         D:\cpphtp4_examples\ch07\Fig07_12\Fig07_12.cpp(31) :
            see declaration of 'x'
```

```
47       cannotSetX( counter, 3 ); // cannotSetX is not a friend
48
49       return 0;
50
51    } // end main
```

Attempting to modify **`private`** data member from non-**`friend`** function results in error.

Using the `this` Pointer

- **`this`** pointer
  - Allows object to access own address
  - Not part of object itself
    - Implicit argument to non-**`static`** member function call
  - Implicitly reference member data and functions
- Type of **`this`** pointer depends on
  - Type of object
  - Whether member function is **`const`**
  - In non-**`const`** member function of **`Employee`**
    - **`this`** has type **`Employee * const`**

    Constant pointer to non-constant **`Employee`** object
  - In **`const`** member function of **`Employee`**
    - **`this`** has type **`const Employee * const`**

    Constant pointer to constant **`Employee`** object

Using the `this` Pointer

```
1       // Fig. 7.13: fig07_13.cpp
2       // Using the this pointer to refer to object members.
3       #include <iostream>
4
5       using std::cout;
6       using std::endl;
7
8       class Test {
9
10    public:
11       Test( int = 0 );    // default constructor
12       void print() const;
13
14    private:
15       int x;
16
17    }; // end class Test
18
19    // constructor
20    Test::Test( int value )
21       : x( value )  // initialize x to value
22    {
23       // empty body
24
25    } // end Test constructor
```

Using the `this` Pointer

```
26
27    // print x using implicit and explicit this poi
28    // parentheses around *this required
29    void Test::print() const
30    {
31       // implicitly use this pointer to access member
32       cout << "        x = " << x;
33
34       // explicitly use this pointer to access member x
35       cout << "\n  this->x = " << this->x;
36
37       // explicitly use dereferenced this pointer and
38       // the dot operator to access member x
39       cout << "\n(*this).x = " << ( *this ).x << endl;
40
41    } // end function print
42
43    int main()
44    {
45       Test testObject( 12 );
46
47       testObject.print();
48
49       return 0;
50
51    } // end main
```

Implicitly use **this** pointer; only specify name of data member (**x**).

Explicitly use **this** pointer with arrow operator

Explicitly use **this** pointer; dereference **this** pointer first, then use dot operator

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

Using the `this` Pointer

Cascaded member function calls

- Multiple functions invoked in same statement
- Function returns reference pointer to same object

```
{ return *this; }
```

- Other functions operate on that pointer
- Functions that do not return references must be called last

Using the `this` Pointer

```
1       // Fig. 7.14: time6.h
2       // Cascading member function calls.
3
4       // Time class definition.
5       // Member functions defined in time6.cpp.
6       #ifndef TIME6_H
7       #define TIME6_H
8
9       class Time {
10
11    public:
12       Time( int = 0, int = 0, int = 0 );
13
14       // set functions
15       Time &setTime( int, int, int ); // set hour, minute, second
16       Time &setHour( int );    // set hour
17       Time &setMinute( int );  // set minute
18       Time &setSecond( int );  // set second
19
20       // get functions (normally declared const)
21       int getHour() const;     // return hour
22       int getMinute() const;   // return minute
23       int getSecond() const;   // return second
24
```

Set functions return reference to **`Time`** object to enable cascaded member function calls.

Using the `this` Pointer

```
25       // print functions (normally declared const)
26       void printUniversal() const;  // print universal time
27       void printStandard() const;   // print standard time
28
29    private:
30       int hour;     // 0 - 23 (24-hour clock format)
31       int minute;   // 0 - 59
32       int second;   // 0 - 59
33
34    }; // end class Time
35
36    #endif
```

Using the `this` Pointer

```
1     // Fig. 7.15: time6.cpp
2     // Member-function definitions for Time class.
3     #include <iostream>
4
5     using std::cout;
6
7     #include <iomanip>
8
9     using std::setfill;
10    using std::setw;
11
12    #include "time6.h"  // Time class definition
13
14    // constructor function to initialize private data;
15    // calls member function setTime to set variables;
16    // default values are 0 (see class definition)
17    Time::Time( int hr, int min, int sec )
18    {
19       setTime( hr, min, sec );
20
21    } // end Time constructor
22
```

Using the `this` Pointer

```
23    // set values of hour, minute, and second
24    Time &Time::setTime( int h, int m, int s )
25    {
26       setHour( h );
27       setMinute( m );
28       setSecond( s );
29
30       return *this;   // enables cascading
31
32    } // end function setTime
33
34    // set hour value
35    Time &Time::setHour( int h )
36    {
37       hour = ( h >= 0 && h < 24 ) ?
38
39       return *this;   // enables cascading
40
41    } // end function setHour
42
```

Return ***this** as reference to enable cascaded member function calls

Return ***this** as reference to enable cascaded member function calls

Using the `this` Pointer

```
43    // set minute value
44    Time &Time::setMinute( int m )
45    {
46       minute = ( m >= 0 && m < 60 ) ? m : 0;
47
48       return *this;    // enables cascading
49
50    } // end function setMinute
51
52    // set second value
53    Time &Time::setSecond( int s )
54    {
55       second = ( s >= 0 && s < 60 ) ? s : 0;
56
57       return *this;    // enables cascading
58
59    } // end function setSecond
60
61    // get hour value
62    int Time::getHour() const
63    {
64       return hour;
65
66    } // end function getHour
67
```

Return ***this** as reference to enable cascaded member function calls

Return ***this** as reference to enable cascaded member function calls

Using the `this` Pointer

```
68    // get minute value
69    int Time::getMinute() const
70    {
71       return minute;
72
73    } // end function getMinute
74
75    // get second value
76    int Time::getSecond() const
77    {
78       return second;
79
80    } // end function getSecond
81
82    // print Time in universal format
83    void Time::printUniversal() const
84    {
85       cout << setfill( '0' ) << setw( 2 ) << hour << ":"
86            << setw( 2 ) << minute << ":"
87            << setw( 2 ) << second;
88
89    } // end function printUniversal
90
91    // print Time in standard format
92    void Time::printStandard() const
93    {
94       cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
95            << ":" << setfill( '0' ) << setw( 2 ) << minute
96            << ":" << setw( 2 ) << second
97            << ( hour < 12 ? " AM" : " PM" );
98
99    } // end function printStandard
```

```
1   // Fig. 7.16: fig07_16.cpp
2   // Cascading member function calls with the this pointer.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include "time6.h"  // Time class definition
9
10  int main()
11  {
12     Time t;
13
14     // cascaded function calls
15     t.setHour( 18 ).setMinute( 30 ).setSe
16
17     // output time in universal and standard formats
18     cout << "Universal time: ";
19     t.printUniversal();
20
21     cout << "\nStandard time: ";
22     t.printStandard();
23
24     cout << "\n\nNew standard time: ";
25
26     // cascaded function calls
27     t.setTime( 20, 20, 20 ).printStandard();
28
29     cout << endl;
30
31     return 0;
32
33  } // end main
```

Cascade member function calls; recall dot operator associates from left to right

Function call to **printStandard** must appear last; **printStandard** does not return reference to **t**

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

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

`static` Class Members

- **`static`** class variable
  - "Class-wide" data
    - Property of class, not specific object of class
  - Efficient when single copy of data is enough
    - Only the **`static`** variable has to be updated
  - May seem like global variables, but have class scope
    - Only accessible to objects of same class
  - Initialized exactly once at file scope
  - Exist even if no objects of class exist
  - Can be **`public`**, **`private`** or **`protected`**

`static` Class Members

Accessing **`static`** class variables

Accessible through any object of class

- **`public static`** variables

Can also be accessed using binary scope resolution operator(**`::`**)

```
Employee::count
```

- **`private static`** variables

When no class member objects exist

Can only be accessed via **`public static`** member function

To call **`public static`** member function combine class name, binary scope resolution operator (**`::`**) and function name

```
Employee::getCount()
```

`static` Class Members

- **`static`** member functions
  - **C**annot access non-**`static`** data or functions

  No **`this`** pointer for **`static`** functions

    - **`static`** data members and **`static`** member functions exist independent of objects

```
1   // Fig. 7.17: employee2.h
2   // Employee class definition.
3   #ifndef EMPLOYEE2_H
4   #define EMPLOYEE2_H
5
6   class Employee {
7
8   public:
9      Employee( const char *, const char * );  // 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 # objects instantiated
16
17  private:
18     char *firstName;
19     char *lastName;
20
21     // static data member
22     static int count;  // number of objects instantiated
23
24  }; // end class Employee
25
26  #endif
```

static member function can only access **static** data members and member functions

static data member is class-wide data

`static` Class Members

```
1  // Fig. 7.18: employee2.cpp
2  // Member-function definitions for class Employee.
3  #include <iostream>
4
5  using std::cout;
6  using std::endl;
7
8  #include <new>           // C++ standard new operator
9  #include <cstring>       // strcpy and strlen prototypes
10
11 #include "employee2.h"   // Employee class
12
13 // define and initialize static data member
14 int Employee::count = 0;
15
16 // define static member function that ret
17 // Employee objects instantiated
18 int Employee::getCount()
19 {
20    return count;
21
22 } // end static function getCount
```

Initialize **`static`** data member exactly once at file scope

`static` member function accesses **`static`** data member **`count`**

`static` Class Members

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

`new` operator dynamically allocates space

Use **`static`** data member to store total **`count`** of employees

`static` Class Members

```
48       delete [] firstName;  // recapture memory
49       delete [] lastName;   // recapture memory
50
51       --count;  // decrement static count of employees
52
53    } // end destructor ~Employee
54
55    // return first name of
56    const char *Employee::getFirstName() const
57    {
58       // const before return type prevents client from modifying
59       // private data; client should copy returned string before
60       // destructor deletes storage to prevent undefined pointer
61       return firstName;
62
63    } // end function getFirstName
64
65    // return last name of employee
66    const char *Employee::getLastName() const
67    {
68       // const before return type prevents client from modifying
69       // private data; client should copy returned string before
70       // destructor deletes storage to prevent undefined pointer
71       return lastName;
72
73    } // end function getLastName
```

Operator **`delete`** deallocates memory

Use **`static`** data member to store total **`count`** of employees

`static` Class Members

```
1      // Fig. 7.19: fig07_19.cpp
2      // Driver to test class Employee.
3      #include <iostream>
4
5      using std::cout;
6      using std::endl;
7
8      #include <new>           // C++ standard new operator
9
10    #include "employee2.h"  // Employee class definition
11
12    int main()
13    {
14       cout << "Number of employees before instantiation is "
15            << Employee::getCount() << endl;   // use class n
16
17       Employee *e1Ptr = new Employee( "Susan", "Baker" );
18       Employee *e2Ptr = new Employee( "Robert", "Jones" );
19
20       cout << "Number of employees after instantiation is "
21            << e1Ptr->getCount();
22
```

new operator dynamically allocates space

static member function can be invoked on any object of class

```
23       cout << "\n\nEmployee 1: "
24            << e1Ptr->getFirstName()
25            << " " << e1Ptr->getLastName()
26            << "\nEmployee 2: "
27            << e2Ptr->getFirstName()
28            << " " << e2Ptr->getLastName() << "
29
30       delete e1Ptr;  // recapture memory
31       e1Ptr = 0;     // disconnect pointer fr
32       delete e2Ptr;  // recapture memory
33       e2Ptr = 0;     // disconnect pointer fr
34
35       cout << "Number of employees after deletion is "
36            << Employee::getCount() << endl;
37
38       return 0;
39
40    } // end main
```

Operator **delete** deallocates memory

static member function invoked using binary scope resolution operator (no existing class objects)

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

Employee 1: Susan Baker
Employee 2: Robert Jones

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

Operator Overloading

Introduction

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
- Operator sensitive to context
- Examples:
  - `<<` Stream insertion, bitwise left-shift
  - `+` Performs arithmetic on multiple types (integers, floats, etc.)
- Will discuss when to use operator overloading

Fundamentals of Operator Overloading

- Types
  - Built in (**`int`**, **`char`**) or user-defined
  - Can use existing operators with user-defined types
    - Cannot create new operators
- Overloading operators
  - Create a function for the class
  - Name function **`operator`** followed by symbol
    - **`Operator+`** for the addition operator **`+`**

Fundamentals of Operator Overloading

- Using operators on a class object
  - It must be overloaded for that class
    - Exceptions:
    - Assignment operator, **=**
      - Memberwise assignment between objects
    - Address operator, **&**
      - Returns address of object
    - Both can be overloaded
- Overloading provides concise notation

```
object2 = object1.add(object2);
object2 = object2 + object1;
```

Restrictions on Operator Overloading

- Cannot change
  - How operators act on built-in data types
    - I.e., cannot change integer addition
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - **&** is unitary, only acts on one operand
- Cannot create new operators
- Operators must be overloaded explicitly
  - Overloading **+** does not overload **+=**

Restrictions on Operator Overloading

Operators that can be overloaded							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
  - Member functions
    - Use **`this`** keyword to implicitly get argument
    - Gets left operand for binary operators (like **+**)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **`friend`** to access **`private`** or **`protected`** data
  - Called when
    - Left operand of binary operator of same class
    - Single operand of unitary operator of same class

Operator Functions As Class Members Vs. As Friend Functions

- Overloaded **<<** operator
  - Left operand of type **`ostream &`**
    - Such as **`cout`** object in **`cout << classObject`**
  - Similarly, overloaded **>>** needs **`istream &`**
  - Thus, both must be non-member functions

Operator Functions As Class Members Vs. As Friend Functions

- Commutative operators
  - May want **+** to be commutative
    - So both "**`a + b`**" and "**`b + a`**" work
  - Suppose we have two different classes
  - Overloaded operator can only be member function when its class is on left
  - **`HugeIntClass + Long int`**
  - Can be member function
- When other way, need a non-member overload function
  - **`Long int + HugeIntClass`**

Overloading Stream-Insertion and Stream-Extraction Operators

- **<<** and **>>**
  - Already overloaded to process each built-in type
  - Can also process a user-defined class
- Example program
  - Class **`PhoneNumber`**
    - Holds a telephone number
  - Print out formatted number automatically
    - **`(123) 456-7890`**

```
1     // Fig. 8.3: fig08_03.cpp
2     // Overloading the stream-insertion and
3     // stream-extraction operators.
4     #include <iostream>
5
6     using std::cout;
7     using std::cin;
8     using std::endl;
9     using std::ostream;
10    using std::istream;
11
12    #include <iomanip>
13
14    using std::setw;
15
16    // PhoneNumber class definition
17    class PhoneNumber {
18       friend ostream &operator<<( ostream&, const PhoneNumber & );
19       friend istream &operator>>( istream&, PhoneNumber & );
20
21    private:
22       char areaCode[ 4 ];  // 3-digit area code and null
23       char exchange[ 4 ];  // 3-digit exchange and null
24       char line[ 5 ];      // 4-digit line and null
25
26    }; // end class PhoneNumber
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be non-member **friend** functions, since the object of class **Phonenumber** appears on the right of the operator.

```
cin << object
cout >> object
```

```
27
28    // overloaded stream-insertion operator; cannot be
29    // a member function if we would like to invoke it with
30    // cout << somePhoneNumber;
31    ostream &operator<<( ostream &output, const PhoneNumber &num )
32    {
33       output << "(" << num.areaCode << ") "
34              << num.exchange << "-" << num.line;
35
36       return output;     // enables cout << a << b << c;
37
38    } // end function operator<<
39
40    // overloaded stream-extraction operator; cannot be
41    // a member function if we would like to invoke it with
42    // cin >> somePhoneNumber;
43    istream &operator>>( istream &input, PhoneNumber &num )
44    {
45       input.ignore();                      // skip (
46       input >> setw( 4 ) >> num.areaCode; // input area code
47       input.ignore( 2 );                   // skip ) and space
48       input >> setw( 4 ) >> num.exchange; // input exchange
49       input.ignore();                      // skip dash (-)
50       input >> setw( 5 ) >> num.line;     // input line
51
52       return input;      // enables cin >> a >> b >> c;
53
54    } // end function operator>>
```

```
27
28    // overloaded stream-insertion operator; cannot be
29    // a member function if we would like to invoke it with
30    // cout << somePhoneNumber;
31    ostream &operator<<( ostream &output, const PhoneNumber &num )
32    {
33       output << "(" << num.areaCode << ") "
34              << num.exchange << "-" << num.line;
35
36       return output;      // enables cout << a << b << c;
37
38    } // end function operator<<
39
40    // overloaded stream-extractio
41    // a member function if we wou
42    // cin >> somePhoneNumber;
43    istream &operator>>( istream &
44    {
45       input.ignore();                    // skip (
46       input >> setw( 4 ) >> num.area
47       input.ignore( 2 );
48       input >> setw( 4 ) >> num.exchange; // input exchange
49       input.ignore();                    // skip dash (-)
50       input >> setw( 5 ) >> num.line
51
52       return input;       // enables
53
54    } // end function operator>>
```

The expression:
cout << phone;
is interpreted as the function call:
operator<<(cout, phone);

output is an alias for **cout**

This allows objects to be cascaded.
cout << phone1 << phone2;
first calls
operator<<(cout, phone1), and returns **cout**.

Next, **cout << phone2** executes.

ignore() skips specified number of characters from input (1 by default)

Stream manipulator **setw** restricts number of characters read. **setw(4)** allows 3 characters to be read, leaving room for the null character.

```
55
56    int main()
57    {
58       PhoneNumber phone; // create object phone
59
60       cout << "Enter phone number in the form (123) 456-7890:\n";
61
62       // cin >> phone invokes operator>> by implicitly issuing
63       // the non-member function call operator>>( cin, phone )
64       cin >> phone;
65
66       cout << "The phone number entered was: " ;
67
68       // cout << phone invokes operator<< by implicitly issuing
69       // the non-member function call operator<<( cout, phone )
70       cout << phone << endl;
71
72       return 0;
73
74    } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Overloading Unary Operators

- Overloading unary operators
  - Non-**`static`** member function, no arguments
  - Non-member function, one argument
    - Argument must be class object or reference to class object
  - Remember, **`static`** functions only access **`static`** data

Overloading Unary Operators

- Upcoming example
  - Overload **!** to test for empty string
  - If non-**static** member function, needs no arguments
    - **!s** becomes **s.operator!()**

```
class String {
public:
   bool operator!() const;
   ...
};
```

  - If non-member function, needs one argument
    - **s!** becomes **operator!(s)**

```
class String {
  friend bool operator!( const String & )
  ...
}
```

Overloading Binary Operators

- Overloading binary operators
  - Non-**`static`** member function, one argument
  - Non-member function, two arguments
- One argument must be class object or reference
  - Upcoming example
  - If non-**`static`** member function, needs one argument

```
class String {
  public:
    const String &operator+=( const String & );
     …
    };
```

  - **`y += z`** equivalent to **`y.operator+=( z )`**

Overloading Binary Operators

- Upcoming example
- If non-member function, needs two arguments
- Example:

```
class String {
   friend const String &operator+=(
                  String &, const String & );
   ...
};
```

  - **`y += z`** equivalent to **`operator+=( y, z )`**

Case Study: Array class

- Arrays in C++
  - No range checking
  - Cannot be compared meaningfully with **==**
  - No array assignment (array names **const** pointers)
  - Cannot input/output entire arrays at once
    - One element at a time
- Example:Implement an **Array** class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with **<<** and **>>**
  - Array comparisons with **==** and **!=**

Case Study: Array class

- Copy constructor
  - Used whenever copy of object needed
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another

      ```
      Array newArray( oldArray );
      ```

      newArray copy of **oldArray**
- Prototype for class **Array**
  - **Array(const Array &);**
  - *Must* take reference
    - Otherwise, pass by value
    - Tries to make copy by calling copy constructor…
    - Infinite loop

Array class

```
1       // Fig. 8.4: array1.h
2       // Array class for storing arrays of integers.
3       #ifndef ARRAY1_H
4       #define ARRAY1_H
5
6       #include <iostream>
7
8       using std::ostream;
9       using std::istream;
10
11    class Array {
12       friend ostream &operator<<( ostream &, const Array & );
13       friend istream &operator>>( istream &, Array & );
14
15    public:
16       Array( int = 10 );          // default co
17       Array( const Array & );     // copy constructor
18       ~Array();                   // destructor
19       int getSize() const;        // return size
20
21       // assignment operator
22       const Array &operator=( const Array & );
23
24       // equality operator
25       bool operator==( const Array & ) const;
26
```

Most operators overloaded as member functions (except **<<** and **>>**, which must be non-member functions).

Prototype for copy constructor

Array class

```
27       // inequality operator; returns opposite of == operator
28       bool operator!=( const Array &right ) const
29       {
30          return ! ( *this == right ); // invokes Array::operator==
31
32       } // end function operator!=
33
34       // subscript operator for non-const objects returns lvalue
35       int &operator[]( int );
36
37       // subscript operator for const objects returns rvalue
38       const int &operator[]( int ) const;
39
40    private:
41       int size; // array size
42       int *ptr; // pointer to first element of array
43
44    }; // end class Array
45
46    #endif
```

!= operator simply returns opposite of **==** operator. Thus, only need to define the **==** operator.

Array class

```
1     // Fig 8.5: array1.cpp
2     // Member function definitions for class Array
3     #include <iostream>
4
5     using std::cout;
6     using std::cin;
7     using std::endl;
8
9     #include <iomanip>
10
11    using std::setw;
12
13    #include <new>        // C++ standard "new" operator
14
15    #include <cstdlib>    // exit function prototype
16
17    #include "array1.h"   // Array class definition
18
19    // default constructor for class Array (default size 10)
20    Array::Array( int arraySize )
21    {
22       // validate arraySize
23       size = ( arraySize > 0 ? arraySize : 10 );
24
25       ptr = new int[ size ]; // create space for array
26
```

Array class

```
27       for ( int i = 0; i < size; i++ )
28          ptr[ i ] = 0;           // initialize array
29
30    } // end Array default constructor
31
32    // copy constructor for class Array;
33    // must receive a reference to prevent
34    Array::Array( const Array &arrayToCopy
35       : size( arrayToCopy.size )
36    {
37       ptr = new int[ size ]; // create space for array
38
39       for ( int i = 0; i < size; i++ )
40          ptr[ i ] = arrayToCopy.ptr[ i ];  // copy into object
41
42    } // end Array copy constructor
43
44    // destructor for class Array
45    Array::~Array()
46    {
47       delete [] ptr;  // reclaim array space
48
49    } // end destructor
50
```

We must declare a new integer array so the objects do not point to the same memory.

Array class

```
51    // return size of array
52    int Array::getSize() const
53    {
54       return size;
55
56    } // end function getSize
57
58    // overloaded assignment operato
59    // const return avoids: ( a1 = a2 ) = a3
60    const Array &Array::operator=( const Array &right )
61    {
62       if ( &right != this ) {  // check for self-assignment
63
64          // for arrays of different sizes, deallocate original
65          // left-side array, then allocate new left-side array
66          if ( size != right.size ) {
67             delete [] ptr;          // reclaim space
68             size = right.size;      // resize this object
69             ptr = new int[ size ]; // create space for array copy
70
71          } // end inner if
72
73          for ( int i = 0; i < size; i++ )
74             ptr[ i ] = right.ptr[ i ];  // copy array into object
75
76       } // end outer if
```

Want to avoid self-assignment.

Array class

```
77
78       return *this;   // enables x = y = z, for example
79
80    } // end function operator=
81
82    // determine if two arrays are equal and
83    // return true, otherwise return false
84    bool Array::operator==( const Array &right ) const
85    {
86       if ( size != right.size )
87          return false;    // arrays of different sizes
88
89       for ( int i = 0; i < size; i++ )
90
91          if ( ptr[ i ] != right.ptr[ i ] )
92             return false; // arrays are not equal
93
94       return true;         // arrays are equal
95
96    } // end function operator==
97
```

Array class

```
98    // overloaded subscript operator for non-const Arrays
99    // reference return creates an lvalue
100 int &Array::operator[]( int subscript )
101 {
102    // check for subscript out of range error
103    if ( subscript < 0 || subscript >= size ) {
104       cout << "\nError: Subscript " << subscript
105            << " out of range" << endl;
106
107       exit( 1 );  // terminate program; subscript out of range
108
109    } // end if
110
111    return ptr[ subscript ]; // reference return
112
113 } // end function operator[]
114
```

integers1[5] calls **integers1.operator[](5)**

exit() (header **<cstdlib>**) ends the program.

```
115 // overloaded subscript operator for const Arrays
116 // const reference return creates an rvalue
117 const int &Array::operator[]( int subscript ) const
118 {
119    // check for subscript out of range error
120    if ( subscript < 0 || subscript >= size ) {
121       cout << "\nError: Subscript " << subscript
122            << " out of range" << endl;
123
124       exit( 1 );  // terminate program; subscript out of range
125
126    } // end if
127
128    return ptr[ subscript ]; // const reference return
129
130 } // end function operator[]
131
132 // overloaded input operator for class Array;
133 // inputs values for entire array
134 istream &operator>>( istream &input, Array &a )
135 {
136    for ( int i = 0; i < a.size; i++ )
137       input >> a.ptr[ i ];
138
139    return input;   // enables cin >> x >> y;
140
141 } // end function
```

Array class

```
142
143 // overloaded output operator for class Array
144 ostream &operator<<( ostream &output, const Array &a )
145 {
146    int i;
147
148    // output private ptr-based array
149    for ( i = 0; i < a.size; i++ ) {
150       output << setw( 12 ) << a.ptr[ i ];
151
152       if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
153          output << endl;
154
155    } // end for
156
157    if ( i % 4 != 0 )  // end last line of output
158       output << endl;
159
160    return output;   // enables cout << x << y;
161
162 } // end function operator<<
```

Array class

```
1     // Fig. 8.6: fig08_06.cpp
2     // Array class test program.
3     #include <iostream>
4
5     using std::cout;
6     using std::cin;
7     using std::endl;
8
9     #include "array1.h"
10
11    int main()
12    {
13       Array integers1( 7 );  // seven-element Array
14       Array integers2;       // 10-element Array by default
15
16       // print integers1 size and contents
17       cout << "Size of array integers1 is "
18            << integers1.getSize()
19            << "\nArray after initialization:\n" << integers1;
20
21       // print integers2 size and contents
22       cout << "\nSize of array integers2 is "
23            << integers2.getSize()
24            << "\nArray after initialization:\n" << integers2;
25
```

Array class

```
26       // input and print integers1 and integers2
27       cout << "\nInput 17 integers:\n";
28       cin >> integers1 >> integers2;
29
30       cout << "\nAfter input, the arrays contain:\n"
31            << "integers1:\n" << integers1
32            << "integers2:\n" << integers2;
33
34       // use overloaded inequality (!=) operator
35       cout << "\nEvaluating: integers1 != integers2\n";
36
37       if ( integers1 != integers2 )
38          cout << "integers1 and integers2 are not equal\n";
39
40       // create array integers3 using integers1 as an
41       // initializer; print size and contents
42       Array integers3( integers1 );  // calls copy constructor
43
44       cout << "\nSize of array integers3 is "
45            << integers3.getSize()
46            << "\nArray after initialization:\n" << integers3;
47
```

```
48       // use overloaded assignment (=) operator
49       cout << "\nAssigning integers2 to integers1:\n";
50       integers1 = integers2;  // note target is smaller
51
52       cout << "integers1:\n" << integers1
53            << "integers2:\n" << integers2;
54
55       // use overloaded equality (==) operator
56       cout << "\nEvaluating: integers1 == integers2\n";
57
58       if ( integers1 == integers2 )
59          cout << "integers1 and integers2 are equal\n";
60
61       // use overloaded subscript operator to create rvalue
62       cout << "\nintegers1[5] is " << integers1[ 5 ];
63
64       // use overloaded subscript operator to create lvalue
65       cout << "\n\nAssigning 1000 to integers1[5]\n";
66       integers1[ 5 ] = 1000;
67       cout << "integers1:\n" << integers1;
68
69       // attempt to use out-of-range subscript
70       cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
71       integers1[ 15 ] = 1000;  // ERROR: out of range
72
73       return 0;
74
75    } // end main
```

```
Size of array integers1 is 7
Array after initialization:
           0           0           0           0
           0           0           0

Size of array integers2 is 10
Array after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the arrays contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
```

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of array integers3 is 7
Array after initialization:
           1           2           3           4
           5           6           7

Assigning integers2 to integers1:
integers1:
           8           9          10          11
          12          13          14          15
          16          17
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13
```

```
Assigning 1000 to integers1[5]
integers1:
           8            9           10           11
          12         1000           14           15
          16           17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range
```

Converting between Types

- Casting
  - Traditionally, cast integers to floats, etc.
  - May need to convert between user-defined types
- Cast operator (conversion operator)
  - Convert from
    - One class to another
    - Class to built-in type (**`int`**, **`char`**, etc.)
  - Must be non-**`static`** member function
    - Cannot be **`friend`**
  - Do not specify return type
    - Implicitly returns type to which you are converting

Converting between Types

- Example
  - Prototype

    `A::operator char *() const;`

    - Casts class **`A`** to a temporary **`char *`**
    - **`(char *)s`** calls **`s.operator char*()`**
- Also
  - **`A::operator int() const;`**
  - **`A::operator OtherClass() const;`**

Converting between Types

- Casting can prevent need for overloading
  - Suppose class **`String`** can be cast to **`char *`**
  - **`cout << s; // s is a String`**
    - Compiler implicitly converts **`s`** to **`char *`**
    - Do not have to overload **`<<`**
  - Compiler can only do 1 cast

Case Study: A `String` Class

- Build class **`String`**
  - String creation, manipulation
  - Class **`string`** in standard library
- Conversion constructor
  - Single-argument constructor
  - Turns objects of other types into class objects
    - **`String s1("hi");`**
    - Creates a **`String`** from a **`char *`**
  - Any single-argument constructor is a conversion constructor

String Class

```
1     // Fig. 8.7: string1.h
2     // String class definition.
3     #ifndef STRING1_H
4     #define STRING1_H
5
6     #include <iostream>
7
8     using std::ostream;
9     using std::istream;
10
11    class String {
12       friend ostream &operator<<( ostrea
13       friend istream &operator>>( istrea
14
15    public:
16       String( const char * = "" ); // conversion/default ctor
17       String( const String & );    // copy constructor
18       ~String();                   // destructor
19
20       const String &operator=( const String & );
21       const String &operator+=( const String & );
22
23       bool operator!() const;                    //
24       bool operator==( const String & ) const; //
25       bool operator<( const String & ) const;  //
26
```

Conversion constructor to make a **String** from a **char *.**

s1 += s2 interpreted as **s1.operator+=(s2)**

Can also concatenate a **String** and a **char *** because the compiler will cast the **char *** argument to a **String**. However, it can only do 1 level of casting.

```
27       // test s1 != s2
28       bool operator!=( const String & right ) const
29       {
30          return !( *this == right );
31
32       } // end function operator!=
33
34       // test s1 > s2
35       bool operator>( const String &right ) const
36       {
37          return right < *this;
38
39       } // end function operator>
40
41       // test s1 <= s2
42       bool operator<=( const String &right ) const
43       {
44          return !( right < *this );
45
46       } // end function operator <=
47
48       // test s1 >= s2
49       bool operator>=( const String &right ) const
50       {
51          return !( *this < right );
52
53       } // end function operator>=
```

String Class

```
54
55       char &operator[]( int );                //
56       const char &operator[]( int ) const; //
57
58       String operator()( int, int );          // return a substring
59
60       int getLength() const;                  //
61
62    private:
63       int length;  // string length
64       char *sPtr;  // pointer to start of string
65
66       void setString( const char * );  // utility function
67
68    }; // end class String
69
70    #endif
```

Two overloaded subscript operators, for **const** and non-**const** objects.

Overload the function call operator **()** to return a substring. This operator can have any amount of operands.

String Class

```
1   // Fig. 8.8: string1.cpp
2   // Member function definitions for class String.
3   #include <iostream>
4
5   using std::cout;
6   using std::endl;
7
8   #include <iomanip>
9
10  using std::setw;
11
12  #include <new>          // C++ standard "new" operator
13
14  #include <cstring>      // strcpy and strcat prototypes
15  #include <cstdlib>      // exit prototype
16
17  #include "string1.h"    // String class definition
18
19  // conversion constructor converts char * to String
20  String::String( const char *s )
21     : length( strlen( s ) )
22  {
23     cout << "Conversion constructor: " << s << '\n';
24     setString( s );          // call utility function
25
26  } // end String conversion constructor
```

```
27
28    // copy constructor
29    String::String( const String &copy )
30       : length( copy.length )
31    {
32       cout << "Copy constructor: " << copy.sPtr << '\n';
33       setString( copy.sPtr ); // call utility function
34
35    } // end String copy constructor
36
37    // destructor
38    String::~String()
39    {
40       cout << "Destructor: " << sPtr << '\n';
41       delete [] sPtr;          // reclaim string
42
43    } // end ~String destructor
44
45    // overloaded = operator; avoids self assignment
46    const String &String::operator=( const String &right )
47    {
48       cout << "operator= called\n";
49
50       if ( &right != this ) {          // avoid self assignment
51          delete [] sPtr;               // prevents memory leak
52          length = right.length;        // new String length
53          setString( right.sPtr );      // call utility function
54       }
```

```
55
56       else
57          cout << "Attempted assignment of a String to itself\n";
58
59       return *this;   // enables cascaded assignments
60
61    } // end function operator=
62
63    // concatenate right operand to this object and
64    // store in this object.
65    const String &String::operator+=( const String &right )
66    {
67       size_t newLength = length + right.length;   // new length
68       char *tempPtr = new char[ newLength + 1 ];  // create memory
69
70       strcpy( tempPtr, sPtr );                    // copy sPtr
71       strcpy( tempPtr + length, right.sPtr );  // copy right.sPtr
72
73       delete [] sPtr;       // reclaim old space
74       sPtr = tempPtr;       // assign new array to sPtr
75       length = newLength;  // assign new length to length
76
77       return *this;  // enables cascaded calls
78
79    } // end function operator+=
80
```

String Class

```
81    // is this String empty?
82    bool String::operator!() const
83    {
84       return length == 0;
85
86    } // end function operator!
87
88    // is this String equal to right String?
89    bool String::operator==( const String &right ) const
90    {
91       return strcmp( sPtr, right.sPtr ) == 0;
92
93    } // end function operator==
94
95    // is this String less than right String?
96    bool String::operator<( const String &right ) const
97    {
98       return strcmp( sPtr, right.sPtr ) < 0;
99
100   } // end function operator<
101
```

```
102 // return reference to character in String as lvalue
103 char &String::operator[]( int subscript )
104 {
105    // test for subscript out of range
106    if ( subscript < 0 || subscript >= length ) {
107       cout << "Error: Subscript " << subscript
108            << " out of range" << endl;
109
110       exit( 1 );  // terminate program
111    }
112
113    return sPtr[ subscript ];  // creates lvalue
114
115 } // end function operator[]
116
117 // return reference to character in String as rvalue
118 const char &String::operator[]( int subscript ) const
119 {
120    // test for subscript out of range
121    if ( subscript < 0 || subscript >= length ) {
122       cout << "Error: Subscript " << subscript
123            << " out of range" << endl;
124
125       exit( 1 );  // terminate program
126    }
127
128    return sPtr[ subscript ];  // creates rvalue
129
130 } // end function operator[]
```

String Class

```
131
132 // return a substring beginning at index and
133 // of length subLength
134 String String::operator()( int index, int subLength )
135 {
136    // if index is out of range or substring length < 0,
137    // return an empty String object
138    if ( index < 0 || index >= length || subLength < 0 )
139       return "";  // converted to a String object automatically
140
141    // determine length of substring
142    int len;
143
144    if ( ( subLength == 0 ) || ( index + subLength > length ) )
145       len = length - index;
146    else
147       len = subLength;
148
149    // allocate temporary array for substring and
150    // terminating null character
151    char *tempPtr = new char[ len + 1 ];
152
153    // copy substring into char array and terminate string
154    strncpy( tempPtr, &sPtr[ index ], len );
155    tempPtr[ len ] = '\0';
```

String Class

```
156
157     // create temporary String object containing the substring
158     String tempString( tempPtr );
159     delete [] tempPtr;  // delete temporary array
160
161     return tempString;  // return copy of the temporary String
162
163  } // end function operator()
164
165  // return string length
166  int String::getLength() const
167  {
168     return length;
169
170  } // end function getLenth
171
172  // utility function called by constructors and operator=
173  void String::setString( const char *string2 )
174  {
175     sPtr = new char[ length + 1 ]; // allocate memory
176     strcpy( sPtr, string2 );       // copy literal to object
177
178  } // end function setString
```

String Class

```
179
180  // overloaded output operator
181  ostream &operator<<( ostream &output, const String &s )
182  {
183     output << s.sPtr;
184
185     return output;   // enables cascading
186
187  } // end function operator<<
188
189  // overloaded input operator
190  istream &operator>>( istream &input, String &s )
191  {
192     char temp[ 100 ];   // buffer to store input
193
194     input >> setw( 100 ) >> temp;
195     s = temp;           // use String class assignment operator
196
197     return input;    // enables cascading
198
199  } // end function operator>>
```

String Class

```
1       // Fig. 8.9: fig08_09.cpp
2       // String class test program.
3       #include <iostream>
4
5       using std::cout;
6       using std::endl;
7
8       #include "string1.h"
9
10      int main()
11      {
12         String s1( "happy" );
13         String s2( " birthday" );
14         String s3;
15
16         // test overloaded equality and relational operators
17         cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18              << "\"; s3 is \"" << s3 << '\"'
19              << "\n\nThe results of comparing s2 and s1:"
20              << "\ns2 == s1 yields "
21              << ( s2 == s1 ? "true" : "false" )
22              << "\ns2 != s1 yields "
23              << ( s2 != s1 ? "true" : "false" )
24              << "\ns2 >  s1 yields "
25              << ( s2 > s1 ? "true" : "false" )
```

String Class

```
26            << "\ns2 <  s1 yields "
27            << ( s2 < s1 ? "true" : "false" )
28            << "\ns2 >= s1 yields "
29            << ( s2 >= s1 ? "true" : "false" )
30            << "\ns2 <= s1 yields "
31            << ( s2 <= s1 ? "true" : "false" );
32
33       // test overloaded String empty (!) operator
34       cout << "\n\nTesting !s3:\n";
35
36       if ( !s3 ) {
37          cout << "s3 is empty; assigning s1 to s3;\n";
38          s3 = s1;  // test overloaded assignment
39          cout << "s3 is \"" << s3 << "\"";
40       }
41
42       // test overloaded String concatenation operator
43       cout << "\n\ns1 += s2 yields s1 = ";
44       s1 += s2;  // test overloaded concatenation
45       cout << s1;
46
47       // test conversion constructor
48       cout << "\n\ns1 += \" to you\" yields\n";
49       s1 += " to you";  // test conversion constructor
50       cout << "s1 = " << s1 << "\n\n";
```

String Class

```
51
52       // test overloaded function call operator () for substring
53       cout << "The substring of s1 starting at\n"
54            << "location 0 for 14 characters, s1(0, 14), is:\n"
55            << s1( 0, 14 ) << "\n\n";
56
57       // test substring "to-end-of-String" option
58       cout << "The substring of s1 starting at\n"
59            << "location 15, s1(15, 0), is: "
60            << s1( 15, 0 ) << "\n\n";  // 0 is "to end of string"
61
62       // test copy constructor
63       String *s4Ptr = new String( s1 );
64       cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
65
66       // test assignment (=) operator with self-assignment
67       cout << "assigning *s4Ptr to *s4Ptr\n";
68       *s4Ptr = *s4Ptr;  // test overloaded assignment
69       cout << "*s4Ptr = " << *s4Ptr << '\n';
70
71       // test destructor
72       delete s4Ptr;
73
```

String Class

```
74       // test using subscript operator to create lvalue
75       s1[ 0 ] = 'H';
76       s1[ 6 ] = 'B';
77       cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
78            << s1 << "\n\n";
79
80       // test subscript out of range
81       cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
82       s1[ 30 ] = 'd';     // ERROR: subscript out of range
83
84       return 0;
85
86    } // end main
```

String Class

```
Conversion constructor: happy
Conversion constructor:  birthday
Conversion constructor:
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion constructor:  to you
Destructor:  to you
  s1 = happy birthday to you
```

The constructor and destructor are called for the temporary **`String`** (converted from the **`char * "to you"`**).

String Class

```
Conversion constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15, 0), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range
```

Overloading ++ and --

- Increment/decrement operators can be overloaded
  - Add 1 to a **`Date`** object, **`d1`**
  - Prototype (member function)
    - **`Date &operator++();`**
    - **`++d1`** same as **`d1.operator++()`**
  - Prototype (non-member)
    - **`Friend Date &operator++( Date &);`**
    - **`++d1`** same as **`operator++( d1 )`**

Overloading ++ and --

- To distinguish pre/post increment
  - Post increment has a dummy parameter
    - **`int`** of **`0`**
  - Prototype (member function)
    - **`Date operator++( int );`**
    - **`d1++`** same as **`d1.operator++( 0 )`**
  - Prototype (non-member)
    - **`friend Date operator++( Data &, int );`**
    - **`d1++`** same as **`operator++( d1, 0 )`**
  - Integer parameter does not have a name
    - Not even in function definition

Overloading ++ and --

- Return values
  - Preincrement
    - Returns by reference (**`Date &`**)
    - lvalue (can be assigned)
  - Postincrement
    - Returns by value
    - Returns temporary object with old value
    - rvalue (cannot be on left side of assignment)
- Decrement operator analogous

Case Study: A `Date` Class

- Example **`Date`** class
  - Overloaded increment operator
    - Change day, month and year
  - Overloaded **`+=`** operator
  - Function to test for leap years
  - Function to determine if day is last of month

```
1   // Fig. 8.10: date1.h
2   // Date class definition.
3   #ifndef DATE1_H
4   #define DATE1_H
5   #include <iostream>
6
7   using std::ostream;
8
9   class Date {
10    friend ostream &operator<<( ostream &, const Date & );
11
12  public:
13     Date( int m = 1, int d = 1, int y = 1
14     void setDate( int, int, int ); // set
15
16     Date &operator++();              // preincrement operator
17     Date operator++( int );          // postincrement operator
18
19     const Date &operator+=( int ); // add days, modify object
20
21     bool leapYear( int ) const;    // is this a leap year?
22     bool endOfMonth( int ) const;  // is this end of month?
23
24  private:
25     int month;
26     int day;
27     int year;
28
29     static const int days[];       // array of days per month
30     void helpIncrement();          // utility function
31
32  }; // end class Date
33
34  #endif
```

Note difference between pre and post increment.

Overloading

```
1      // Fig. 8.11: date1.cpp
2      // Date class member function definitions.
3      #include <iostream>
4      #include "date1.h"
5
6      // initialize static member at file scope;
7      // one class-wide copy
8      const int Date::days[] =
9         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
10
11    // Date constructor
12    Date::Date( int m, int d, int y )
13    {
14       setDate( m, d, y );
15
16    } // end Date constructor
17
18    // set month, day and year
19    void Date::setDate( int mm, int dd, int yy )
20    {
21       month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
22       year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
23
```

```
24       // test for a leap year
25       if ( month == 2 && leapYear( year ) )
26          day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
27       else
28          day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
29
30    } // end function setDate
31
32    // overloaded preincrement operator
33    Date &Date::operator++()
34    {
35       helpIncrement();
36
37       return *this;  // reference return to create an lvalue
38
39    } // end function operator++
40
41    // overloaded postincrement operator;
42    // integer parameter does not have a p
43    Date Date::operator++( int )
44    {
45       Date temp = *this;  // hold current
46       helpIncrement();
47
48       // return unincremented, saved, temporary object
49       return temp;   // value return; not a reference return
50
51    } // end function operator++
```

Postincrement updates object and returns a copy of the original. Do not return a reference to temp, because it is a local variable that will be destroyed.

Also note that the integer parameter does not have a name.

Overloading

```
52
53    // add specified number of days to date
54    const Date &Date::operator+=( int additionalDays )
55    {
56       for ( int i = 0; i < additionalDays; i++ )
57          helpIncrement();
58
59       return *this;    // enables cascading
60
61    } // end function operator+=
62
63    // if the year is a leap year, return true;
64    // otherwise, return false
65    bool Date::leapYear( int testYear ) const
66    {
67       if ( testYear % 400 == 0 ||
68          ( testYear % 100 != 0 && testYear % 4 == 0 ) )
69          return true;   // a leap year
70       else
71          return false;  // not a leap year
72
73    } // end function leapYear
74
```

Overloading

```
75    // determine whether the day is the last day of the month
76    bool Date::endOfMonth( int testDay ) const
77    {
78       if ( month == 2 && leapYear( year ) )
79          return testDay == 29; // last day of Feb. in leap year
80       else
81          return testDay == days[ month ];
82
83    } // end function endOfMonth
84
85    // function to help increment the date
86    void Date::helpIncrement()
87    {
88       // day is not end of month
89       if ( !endOfMonth( day ) )
90          ++day;
91
92       else
93
94          // day is end of month and month < 12
95          if ( month < 12 ) {
96             ++month;
97             day = 1;
98          }
99
```

Overloading

```
100         // last day of year
101         else {
102            ++year;
103            month = 1;
104            day = 1;
105         }
106
107   } // end function helpIncrement
108
109   // overloaded output operator
110   ostream &operator<<( ostream &output, const Date &d )
111   {
112      static char *monthName[ 13 ] = { "", "January",
113         "February", "March", "April", "May", "June",
114         "July", "August", "September", "October",
115         "November", "December" };
116
117      output << monthName[ d.month ] << ' '
118             << d.day << ", " << d.year;
119
120      return output;   // enables cascading
121
122   } // end function operator<<
```

Overloading

```
1      // Fig. 8.12: fig08_12.cpp
2      // Date class test program.
3      #include <iostream>
4
5      using std::cout;
6      using std::endl;
7
8      #include "date1.h"  // Date class definition
9
10    int main()
11    {
12       Date d1;  // defaults to January 1, 1900
13       Date d2( 12, 27, 1992 );
14       Date d3( 0, 99, 8045 );  // invalid date
15
16       cout << "d1 is " << d1 << "\nd2 is " << d2
17            << "\nd3 is " << d3;
18
19       cout << "\n\nd2 += 7 is " << ( d2 += 7 );
20
21       d3.setDate( 2, 28, 1992 );
22       cout << "\n\n  d3 is " << d3;
23       cout << "\n++d3 is " << ++d3;
24
25       Date d4( 7, 13, 2002 );
```

Overloading

```
26
27        cout << "\n\nTesting the preincrement operator:\n"
28             << "  d4 is " << d4 << '\n';
29        cout << "++d4 is " << ++d4 << '\n';
30        cout << "  d4 is " << d4;
31
32        cout << "\n\nTesting the postincrement operator:\n"
33             << "  d4 is " << d4 << '\n';
34        cout << "d4++ is " << d4++ << '\n';
35        cout << "  d4 is " << d4 << endl;
36
37        return 0;
38
39    } // end main
```

Overloading

```
d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

  d3 is February 28, 1992
++d3 is February 29, 1992

Testing the preincrement operator:
  d4 is July 13, 2002
++d4 is July 14, 2002
  d4 is July 14, 2002

Testing the postincrement operator:
  d4 is July 14, 2002
d4++ is July 14, 2002
  d4 is July 15, 2002
```

Standard Library Classes `string` and `vector`

- Classes built into C++
  - Available for anyone to use
    - **`string`**
  - Similar to our **`String`** class
    - **`vector`**
    - Dynamically resizable array
- Redo our **`String`** and **`Array`** examples
  - Use **`string`** and **`vector`**

Standard Library Classes `string` and `vector`

- Class **`string`**
  - Header **`<string>`**, namespace **`std`**
  - Can initialize **`string s1("hi");`**
  - Overloaded **`<<`**
    - **`cout << s1`**
  - Overloaded relational operators
    - **`== != >= > <= <`**
  - Assignment operator =
  - Concatenation (overloaded **`+=`**)

Standard Library Classes `string` and `vector`

- Class **`string`**
  - Substring function **`substr`**
    - **`s1.substr(0, 14);`**
  - Starts at location 0, gets 14 characters
    - **`S1.substr(15)`**
      - Substring beginning at location 15
  - Overloaded **`[]`**
    - Access one character
    - No range checking (if subscript invalid)
  - **`at`** function
    - **`s1.at(10)`**
    - Character at subscript 10
    - Has bounds checking
      - Will end program if invalid

Standard Library Classes

```
1     // Fig. 8.13: fig08_13.cpp
2     // Standard library string class test program.
3     #include <iostream>
4
5     using std::cout;
6     using std::endl;
7
8     #include <string>
9
10    using std::string;
11
12    int main()
13    {
14       string s1( "happy" );
15       string s2( " birthday" );
16       string s3;
17
18       // test overloaded equality and relational operators
19       cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
20            << "\"; s3 is \"" << s3 << '\"'
21            << "\n\nThe results of comparing s2 and s1:"
22            << "\ns2 == s1 yields "
23            << ( s2 == s1 ? "true" : "false" )
24            << "\ns2 != s1 yields "
25            << ( s2 != s1 ? "true" : "false" )
```

Standard Library Classes

```
26            << "\ns2 >  s1 yields "
27            << ( s2 > s1 ? "true" : "false" )
28            << "\ns2 <  s1 yields "
29            << ( s2 < s1 ? "true" : "false" )
30            << "\ns2 >= s1 yields "
31            << ( s2 >= s1 ? "true" : "false" )
32            << "\ns2 <= s1 yields "
33            << ( s2 <= s1 ? "true" : "false" );
34
35       // test string member function empty
36       cout << "\n\nTesting s3.empty():\n";
37
38       if ( s3.empty() ) {
39          cout << "s3 is empty; assigning s1 to s3;\n";
40          s3 = s1;  // assign s1 to s3
41          cout << "s3 is \"" << s3 << "\"";
42       }
43
44       // test overloaded string concatenation operator
45       cout << "\n\ns1 += s2 yields s1 = ";
46       s1 += s2;  // test overloaded concatenation
47       cout << s1;
48
```

Standard Library Classes

```
49       // test overloaded string concatenation operator
50       // with C-style string
51       cout << "\n\ns1 += \" to you\" yields\n";
52       s1 += " to you";
53       cout << "s1 = " << s1 << "\n\n";
54
55       // test string member function substr
56       cout << "The substring of s1 starting at location 0 for\n"
57            << "14 characters, s1.substr(0, 14), is:\n"
58            << s1.substr( 0, 14 ) << "\n\n";
59
60       // test substr "to-end-of-string" option
61       cout << "The substring of s1 starting at\n"
62            << "location 15, s1.substr(15), is:\n"
63            << s1.substr( 15 ) << '\n';
64
65       // test copy constructor
66       string *s4Ptr = new string( s1 );
67       cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
68
69       // test assignment (=) operator with self-assignment
70       cout << "assigning *s4Ptr to *s4Ptr\n";
71       *s4Ptr = *s4Ptr;
72       cout << "*s4Ptr = " << *s4Ptr << '\n';
73
```

Standard Library Classes

```
74        // test destructor
75        delete s4Ptr;
76
77        // test using subscript operator to create lvalue
78        s1[ 0 ] = 'H';
79        s1[ 6 ] = 'B';
80        cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
81             << s1 << "\n\n";
82
83        // test subscript out of range with string member function "at"
84        cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
85        s1.at( 30 ) = 'd';     // ERROR: subscript out of range
86
87        return 0;
88
89     } // end main
```

Standard Library Classes

```
s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
s2 == s1 yields false
s2 != s1 yields true
s2 >  s1 yields false
s2 <  s1 yields true
s2 >= s1 yields false
s2 <= s1 yields true

Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
s1 = happy birthday to you

The substring of s1 starting at location 0 for
14 characters, s1.substr(0, 14), is:
happy birthday
```

Standard Library Classes

```
The substring of s1 starting at
location 15, s1.substr(15), is:
to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
*s4Ptr = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at( 30 ) yields:

abnormal program termination
```

Standard Library Classes `string` and `vector`

- Class **`vector`**
  - Header **`<vector>`**, namespace **`std`**
  - Store any type
    - **`vector< int > myArray(10)`**
  - Function **`size`** (**`myArray.size()`**)
  - Overloaded **`[]`**
    - Get specific element, **`myArray[3]`**
  - Overloaded **`!=`**, **`==`**, and **`=`**
    - Inequality, equality, assignment

Standard Library Classes Class vector

```
1      // Fig. 8.14: fig08_14.cpp
2      // Demonstrating standard library class vector.
3      #include <iostream>
4
5      using std::cout;
6      using std::cin;
7      using std::endl;
8
9      #include <iomanip>
10
11     using std::setw;
12
13     #include <vector>
14
15     using std::vector;
16
17     void outputVector( const vector< int > & );
18     void inputVector( vector< int > & );
19
20     int main()
21     {
22        vector< int > integers1( 7 );   // 7-element vector< int >
23        vector< int > integers2( 10 );  // 10-element vector< int >
24
```

Standard Library Classes Class vector

```
25       // print integers1 size and contents
26       cout << "Size of vector integers1 is "
27            << integers1.size()
28            << "\nvector after initialization:\n";
29       outputVector( integers1 );
30
31       // print integers2 size and contents
32       cout << "\nSize of vector integers2 is "
33            << integers2.size()
34            << "\nvector after initialization:\n";
35       outputVector( integers2 );
36
37       // input and print integers1 and integers2
38       cout << "\nInput 17 integers:\n";
39       inputVector( integers1 );
40       inputVector( integers2 );
41
42       cout << "\nAfter input, the vectors contain:\n"
43            << "integers1:\n";
44       outputVector( integers1 );
45       cout << "integers2:\n";
46       outputVector( integers2 );
47
48       // use overloaded inequality (!=) operator
49       cout << "\nEvaluating: integers1 != integers2\n";
50
```

Standard Library Classes Class vector

```
51       if ( integers1 != integers2 )
52          cout << "integers1 and integers2 are not equal\n";
53
54       // create vector integers3 using integers1 as an
55       // initializer; print size and contents
56       vector< int > integers3( integers1 );  // copy constructor
57
58       cout << "\nSize of vector integers3 is "
59            << integers3.size()
60            << "\nvector after initialization:\n";
61       outputVector( integers3 );
62
63
64       // use overloaded assignment (=) operator
65       cout << "\nAssigning integers2 to integers1:\n";
66       integers1 = integers2;
67
68       cout << "integers1:\n";
69       outputVector( integers1 );
70       cout << "integers2:\n";
71       outputVector( integers1 );
72
```

Standard Library Classes Class vector

```
73       // use overloaded equality (==) operator
74       cout << "\nEvaluating: integers1 == integers2\n";
75
76       if ( integers1 == integers2 )
77          cout << "integers1 and integers2 are equal\n";
78
79       // use overloaded subscript operator to create rvalue
80       cout << "\nintegers1[5] is " << integers1[ 5 ];
81
82       // use overloaded subscript operator to create lvalue
83       cout << "\n\nAssigning 1000 to integers1[5]\n";
84       integers1[ 5 ] = 1000;
85       cout << "integers1:\n";
86       outputVector( integers1 );
87
88       // attempt to use out of range subscript
89       cout << "\nAttempt to assign 1000 to integers1.at( 15 )"
90            << endl;
91       integers1.at( 15 ) = 1000;  // ERROR: out of range
92
93       return 0;
94
95    } // end main
96
```

Standard Library Classes Class vector

```
97    // output vector contents
98    void outputVector( const vector< int > &array )
99    {
100     for ( int i = 0; i < array.size(); i++ ) {
101        cout << setw( 12 ) << array[ i ];
102
103        if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
104           cout << endl;
105
106     } // end for
107
108     if ( i % 4 != 0 )
109        cout << endl;
110
111   } // end function outputVector
112
113   // input vector contents
114   void inputVector( vector< int > &array )
115   {
116      for ( int i = 0; i < array.size(); i++ )
117         cin >> array[ i ];
118
119   } // end function inputVector
```

Standard Library Classes Class vector

```
Size of vector integers1 is 7
vector after initialization:
           0           0           0           0
           0           0           0

Size of vector integers2 is 10
vector after initialization:
           0           0           0           0
           0           0           0           0
           0           0

Input 17 integers:
  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
           1           2           3           4
           5           6           7
integers2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Standard Library Classes Class vector

```
Size of vector integers3 is 7
vector after initialization:
           1            2            3            4
           5            6            7

Assigning integers2 to integers1:
integers1:
           8            9           10           11
          12           13           14           15
          16           17
integers2:
           8            9           10           11
          12           13           14           15
          16           17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
           8            9           10           11
          12         1000           14           15
          16           17

Attempt to assign 1000 to integers1.at( 15 )

abnormal program termination
```

Esercitazione 7

Exercise 6

- Write a "Punto" class in 3 dimensions and a program that uses it.
- The data members are the 3 coordinates of the "point".
- There must be functions: `getX()`, `setX(double)`, `getY()`, `setY(double)`, `getZ()`, `setZ(double)`, `print()`, `distance(Point &)`
- The function `distance(Point &)` belongs to an object of the Point class, let's call it `p1`, and its argument is another object of the class Point, `p2`. By writing **`double dist = p1.distance(p2)`** the function must return the distance between `p1` and `p2`.

```
Execution example:
./usaPunto.exe
  Point p1: [1, 2, 3]
  Point p2: [4, 5, 6]
  The distance between p1 and p2: 5.19615
  We access the points via pointer
  Point p1: [1, 2, 3]
  Point p2: [4, 5, 6]
  The distance between p1 and p2: 5.19615
```

Esercizio 6

Declaration: `double distanza(Punto&);`

Definition:

```
double Punto::distanza(Punto& p2)
{
  return sqrt( (x - p2.x)*(x - p2.x) + (y - p2.y)*(y - p2.y) +(z - p2.z)*(z - p2.z) );
}
```

Exrcise 6

```
// Test classe Sfera

#include <iostream>
using std::cout;
using std::endl;

#include "Punto.h"

int main ()
{
  Punto p1(1., 2., 3.);
  Punto p2(4., 5., 6.);

  Punto *p1Ptr = &p1;
  Punto *p2Ptr = &p2;

  cout << " Punto p1: ";
  p1.print();
  cout << " Punto p2: ";
  p2.print();
  cout << " La distanza tra p1 e p2: " << p1.distanza(p2) << endl;

  cout << endl << " Accediamo ai punti via puntatore " << endl;

  cout << " Punto p1: ";
  p1Ptr->print();
  cout << " Punto p2: ";
  p2Ptr->print();
  cout << " La distanza tra p1 e p2: " << p1Ptr->distanza(*p2Ptr) <<
endl;

  return 0;
}
```

Esercitazione 7

Exercise 7

- Implement a "Sfera" class whose data members are:
  - the Center, an object of the Punto class, and the radius (a double)
- There must be:
  - a constructor with arguments "Punto" and a double
  - `set` and `get` functions for the datamembers "center" (i.e. Point) and "radius"
  - functions `getArea()`, `getVolume()`, `getName()`, `print()`
  - a **bool function** that returns true if two spheres overlap, false if they do not overlap (N.B .: two spheres overlap if the distance between the centers is less than the sum of their radii)
- Hint: the Point class can be reused as equal. Start from the class "Sfera" and make the (few) necessary changes.

```
Execution example:
./usaSfera.exe
 Sfera1
 center: [1, 2, 3]
 radius: 4; Surface area: 201.024; Volume: 268.032
 Sfera2
 center: [10, 20, 30]
 radius: 40; Surface area: 20102.4; Volume: 268032
 the spheres overlap
```

Esercitazione 8

Exercise 1

Implement a "Complex" class, which represents complex numbers. The data members are the real and imaginary part of the number.

- There must be a constructor composed by 2 double.
- Overload the operators +,-, *, =, ==,! =, <<, >>

N. B .: the functions that overload the operators << and >> must be declares as "friend" function

```
Execution example:

./usaComplessi.exe

Give me a complex number in the form: a + bi

3 + 5i

x: 0 + 0i

y: 4.3 + 8.2i

z: 3.3 + 1.1i

k: 3 + 5i

x = y + z:

7.6 + 9.3i = 4.3 + 8.2i + 3.3 + 1.1i

x = y - z:

1 + 7.1i = 4.3 + 8.2i - 3.3 + 1.1i

x = y * z:

23.21 + 31.79i = 4.3 + 8.2i * 3.3 + 1.1i

23.21 + 31.79i ! = 3 + 5i

3 + 5i == 3 + 5i
```

Esercitazione 8

Exercise 2

Implement a Vector class that represents vectors in 3 dimensions

(hint: use 3-component representation)

The class must have at least:

- a constructor with 3 double
- overloading of the operators <<, +,-
- the operation "scalar product"
- member functions that return the components and the form of the vector

Write a simple program that creates and uses objects of the Vector class

```
Execution example:

./useVettore.exe
 Vector v1 (1, 0, 0)
 Vector v2 (1, 1, 0)

 Vector v (2, 1, 0) sum of v1 vectors (1,
0, 0) and v2 (1, 1, 0)
 component x: 2
 component y: 1
 component z: 0
 form of v: 2.23607

 The scalar product of (1, 0, 0) and v2
(1, 1, 0): 1
```

Backup

Dynamic Memory Management with Operators `new` and `delete`

- Dynamic memory management
  - Control allocation and deallocation of memory
  - Operators **`new`** and **`delete`**
    - Include standard header **`<new>`**
      - Access to standard version of **`new`**

Dynamic Memory Management with Operators `new` and `delete`

- **`new`**

  Consider

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

  - **`new`** operator
    - Creates object of proper size for type **`Time`**
      - Error if no space in memory for object
    - Calls default constructor for object
    - Returns pointer of specified type

  Providing initializers

```
double *ptr = new double( 3.14159 );
Time *timePtr = new Time( 12, 0, 0 );
```

  Allocating arrays

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

Dynamic Memory Management with Operators `new` and `delete`

- **`delete`**

  Destroy dynamically allocated object and free space

  Consider

  ```
  delete timePtr;
  ```

  Operator **`delete`**

  - Calls destructor for object
  - Deallocates memory associated with object
    - Memory can be reused to allocate other objects

  Deallocating arrays

  ```
  delete [] gradesArray;
  ```

  - Deallocates array to which **`gradesArray`** points
  - If pointer to array of objects
    - First calls destructor for each object in array
    - Then deallocates memory