Operator Overloading

Operator Overloading

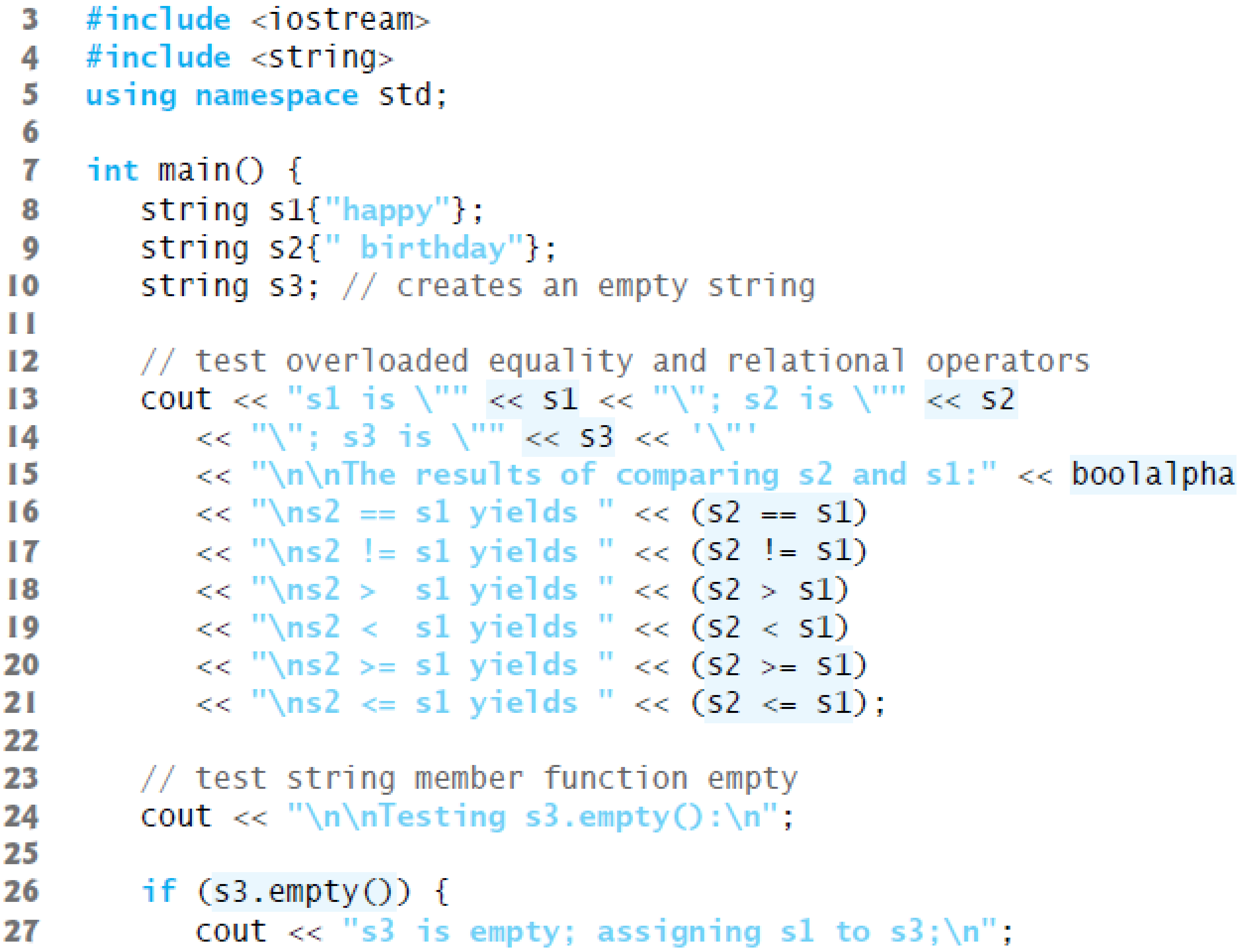
The process when C++ operators work with class objects is supported via **operator overloading**.

One example of an overloaded operator built into C++ is **<<**,
which is used ***both*** as the stream insertion operator *and* as the bitwise left-shift operator **depending on the object type**:

```
int main()
{
   int x = 4;
   x = x << 1;
   cout << x << endl; // prints 8
   return 0;
}
```

- Since ‘x’ is an ‘int’, bitwise left-shift operation is performed for ‘x << 1’.
- Since ‘cout’ is an ‘std::ostream’ object, stream insertion is performed for ‘cout << x’.

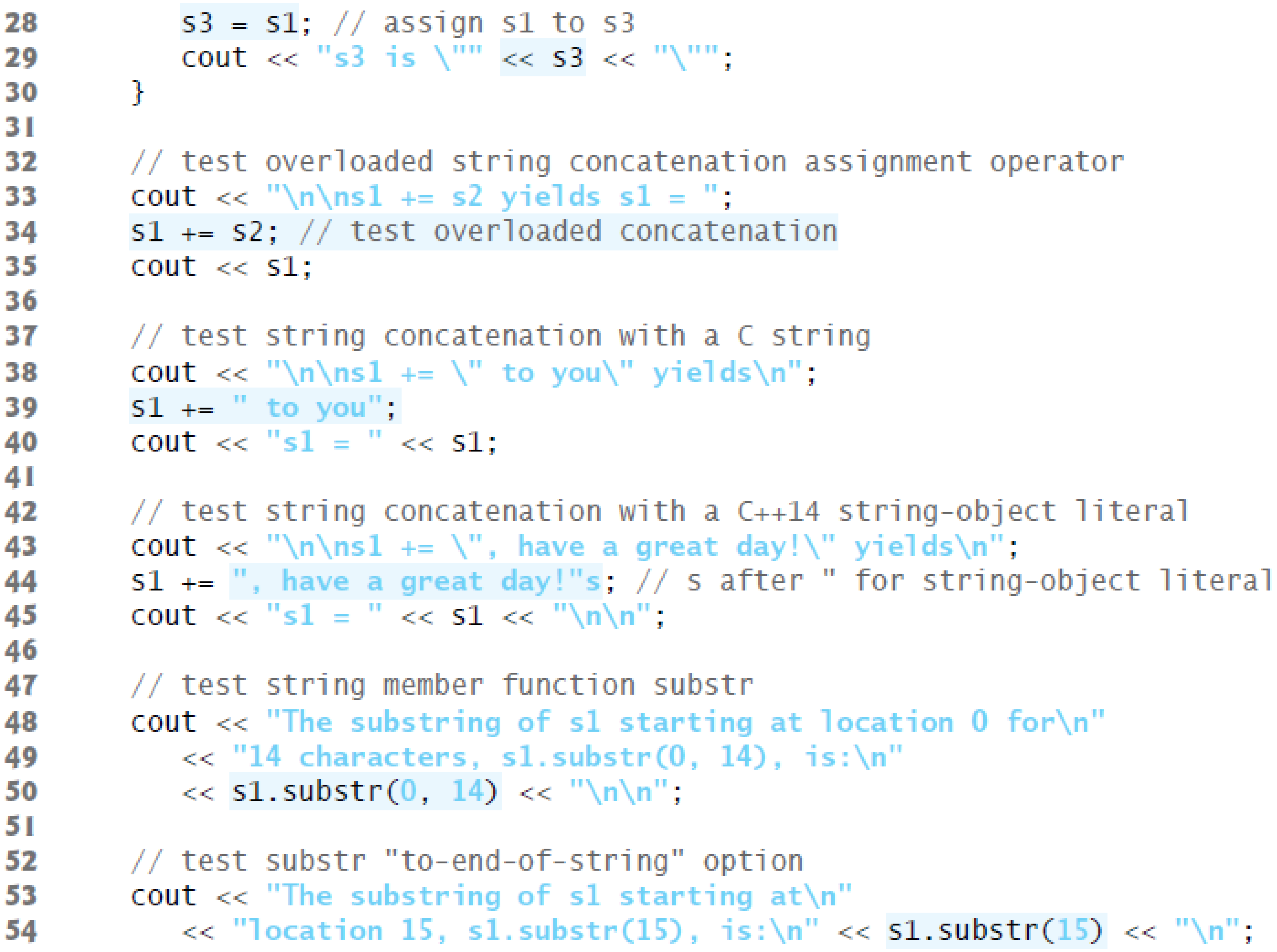
Operator Overloading (std::string)



```
 3  #include <iostream>
 4  #include <string>
 5  using namespace std;
 6
 7  int main() {
 8     string s1{"happy"};
 9     string s2{" birthday"};
10     string s3; // creates an empty string
11
12     // test overloaded equality and relational operators
13     cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
14        << "\"; s3 is \"" << s3 << '\"'
15        << "\n\nThe results of comparing s2 and s1:" << boolalpha
16        << "\ns2 == s1 yields " << (s2 == s1)
17        << "\ns2 != s1 yields " << (s2 != s1)
18        << "\ns2 >  s1 yields " << (s2 > s1)
19        << "\ns2 <  s1 yields " << (s2 < s1)
20        << "\ns2 >= s1 yields " << (s2 >= s1)
21        << "\ns2 <= s1 yields " << (s2 <= s1);
22
23     // test string member function empty
24     cout << "\n\nTesting s3.empty():\n";
25
26     if (s3.empty()) {
27        cout << "s3 is empty; assigning s1 to s3;\n";
```

- Lines 15–21 perform lexicographical comparisons (like a dictionary ordering)
- Stream manipulator **boolalpha** (line 15) to set the output stream to display bool values as the strings "false" and "true".

Operator Overloading (std::string)



```
      s3 = s1; // assign s1 to s3
      cout << "s3 is \"" << s3 << "\"";
   }

   // test overloaded string concatenation assignment operator
   cout << "\n\ns1 += s2 yields s1 = ";
   s1 += s2; // test overloaded concatenation
   cout << s1;

   // test string concatenation with a C string
   cout << "\n\ns1 += \" to you\" yields\n";
   s1 += " to you";
   cout << "s1 = " << s1;

   // test string concatenation with a C++14 string-object literal
   cout << "\n\ns1 += \", have a great day!\" yields\n";
   s1 += ", have a great day!"s; // s after " for string-object literal
   cout << "s1 = " << s1 << "\n\n";

   // test string member function substr
   cout << "The substring of s1 starting at location 0 for\n"
      << "14 characters, s1.substr(0, 14), is:\n"
      << s1.substr(0, 14) << "\n\n";

   // test substr "to-end-of-string" option
   cout << "The substring of s1 starting at\n"
      << "location 15, s1.substr(15), is:\n" << s1.substr(15) << "\n";
```

Operator Overloading (std::string)

```
44| s1 += ", have a great day!"s;
```

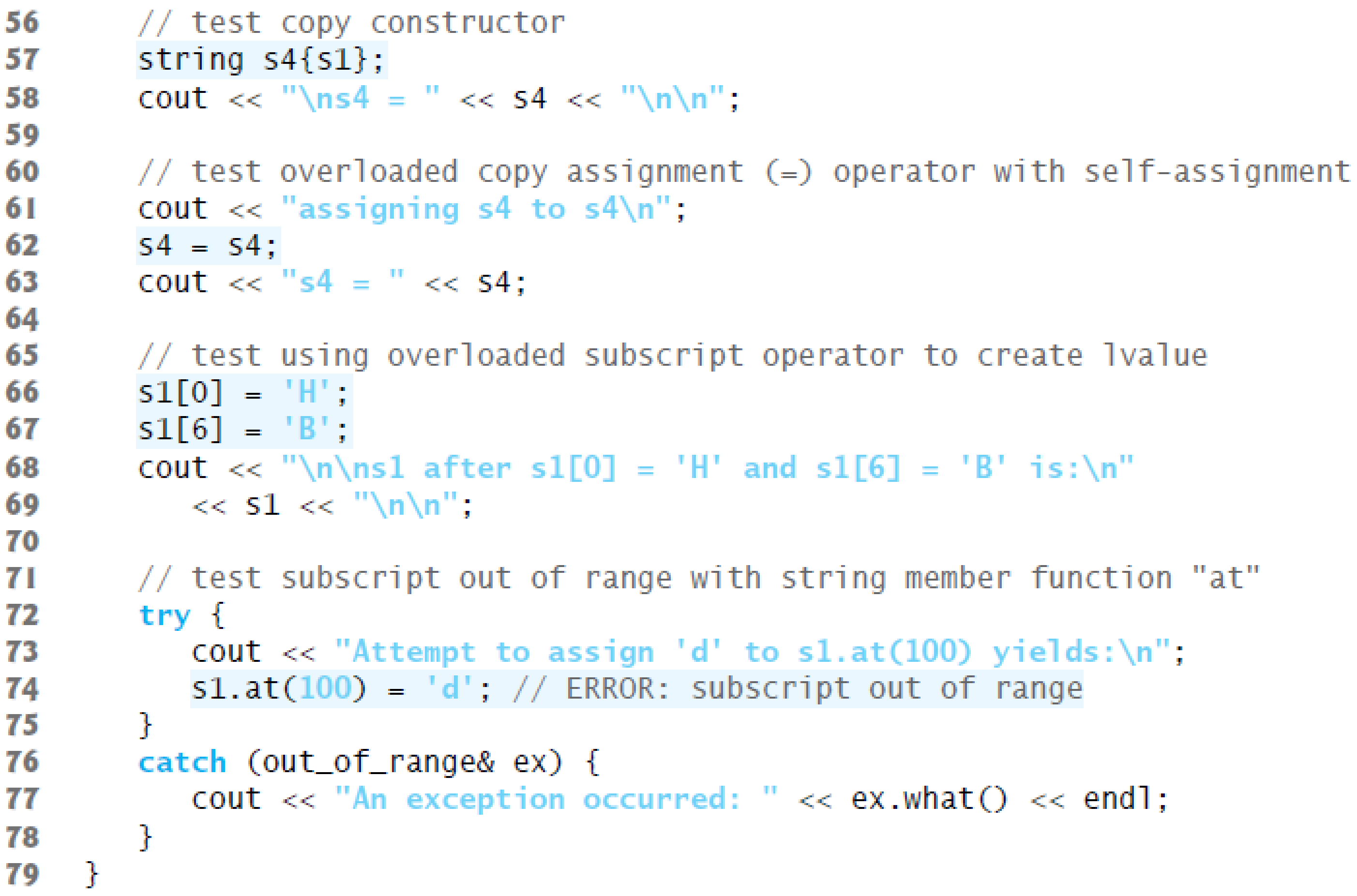
Line 44 concatenates s1 with a **C++14 string-object literal** – “, have a great day!”**s**. The string-object literal actually results in a call to a C++ Standard Library function that returns a **std::string** object containing the literal’s characters.

Example:

cout << "str1" + "str2" << endl; // Error
cout << "str1”s + "str2" << endl; // OK

The compiler converts “str1”s to std::string(“str1”) and the addition operation is performed on the created object.

Operator Overloading (std::string)



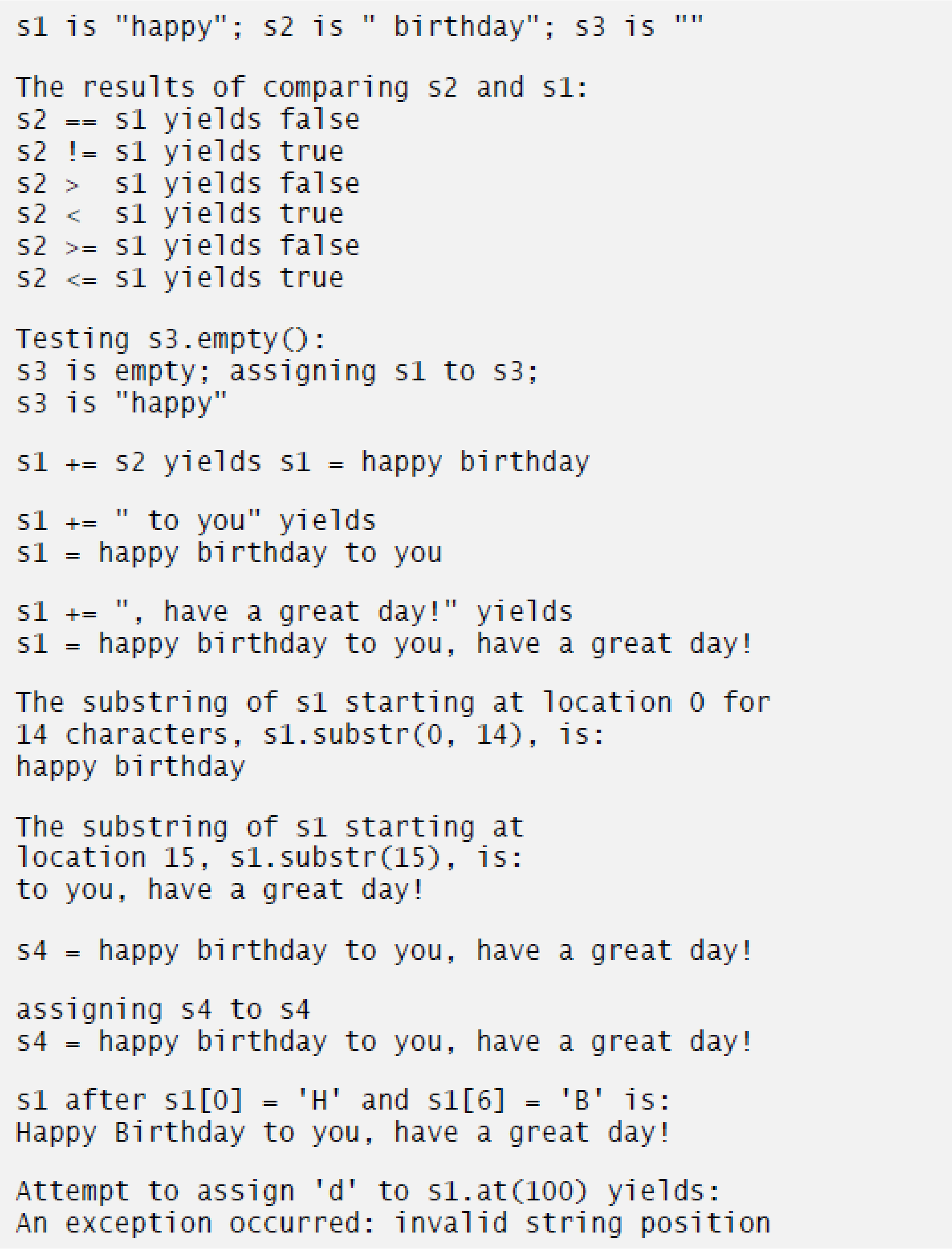
```cpp
   // test copy constructor
   string s4{s1};
   cout << "\ns4 = " << s4 << "\n\n";

   // test overloaded copy assignment (=) operator with self-assignment
   cout << "assigning s4 to s4\n";
   s4 = s4;
   cout << "s4 = " << s4;

   // test using overloaded subscript operator to create lvalue
   s1[0] = 'H';
   s1[6] = 'B';
   cout << "\n\ns1 after s1[0] = 'H' and s1[6] = 'B' is:\n"
      << s1 << "\n\n";

   // test subscript out of range with string member function "at"
   try {
      cout << "Attempt to assign 'd' to s1.at(100) yields:\n";
      s1.at(100) = 'd'; // ERROR: subscript out of range
   }
   catch (out_of_range& ex) {
      cout << "An exception occurred: " << ex.what() << endl;
   }
}
```

Operator Overloading (std::string)

- Class string’s overloaded [] (subscript) operator **does not perform any bounds checking**. Therefore, you must ensure that operations using class string’s overloaded [] operator do not accidentally manipulate elements outside the bounds of the string, thus causing **undefined behavior**.

- Class std::string ***does*** **provide bounds checking** in its member function **at**, which throws an exception if its argument is an *invalid* subscript.

- If the std::string object is const-qualified, the **at** function and **subscript []** operator return a **const char&**. Otherwise, they return a **char&**.



```
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

s1 += ", have a great day!" yields
s1 = happy birthday to you, have a great day!

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

The substring of s1 starting at
location 15, s1.substr(15), is:
to you, have a great day!

s4 = happy birthday to you, have a great day!

assigning s4 to s4
s4 = happy birthday to you, have a great day!

s1 after s1[0] = 'H' and s1[6] = 'B' is:
Happy Birthday to you, have a great day!

Attempt to assign 'd' to s1.at(100) yields:
An exception occurred: invalid string position
```

Fundamentals of Operator Overloading

- *C++* ***does*** **allow** most existing operators to be overloaded so that, when they're used with objects, they have *meaning appropriate to those objects*.

- C++ **does** ***NOT*** **allow** ***new*** operators to be created.

- You must write operator-overloading functions to perform the desired operations.

- An operator is overloaded by writing a **non-static member function definition** or **non-member function definition** as you normally would, except that the **function name starts with the keyword operator** followed by the **symbol** for the operator being overloaded. For example, "operator+", "operator<", etc.

- When operators are overloaded as **member functions**, they must be **non-static**, because *they must be called on an object of the class* and operate on that object.

Fundamentals of Operator Overloading

To use an operator on an object of a class, you must define overloaded operator functions for that class—**with three exceptions (these 3 operators are supported for each class):**

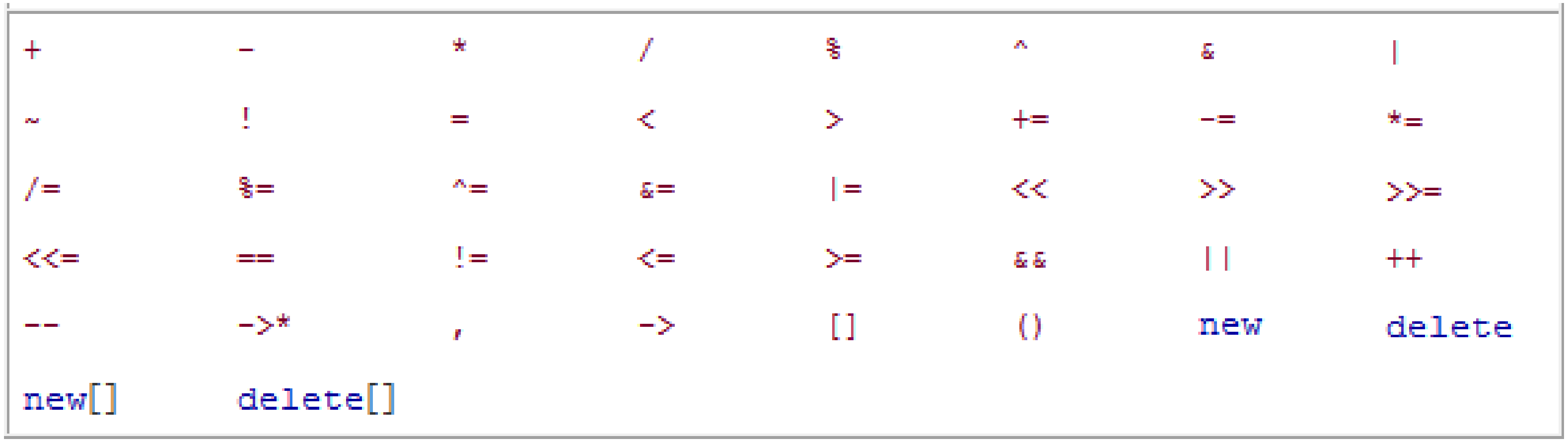
- The ***assignment operator (=)*** may be used with *most* classes to perform *memberwise assignment* of the data members – but memberwise assignment is dangerous for classes with pointer members.

- The ***address (&) operator*** returns a pointer to the object; this operator also can be overloaded.

- The ***comma operator (,)*** evaluates the expression to its left then the expression to its right, and returns the value of the latter expression. This operator also can be overloaded.

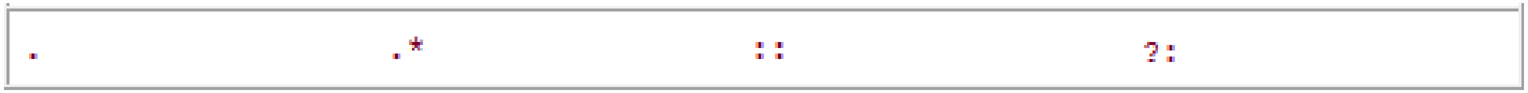
Fundamentals of Operator Overloading

Operators that can be overloaded



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

Operators that canNOT be overloaded



.	.*	::	?:

Rules and Restrictions on Operator Overloading

As we prepare to overload operators for our own classes, there are several rules and restrictions you should keep in mind:

- An operator's **precedence cannot be changed by overloading**. Parentheses can be used to force the order of evaluation of overloaded operators in an expression.
- An operator's **associativity cannot be changed by overloading**—if an operator normally associates from left to right, then so do all of its overloaded versions.
- **The number of operands an operator takes cannot be changed**—overloaded unary operators remain unary operators; overloaded binary operators remain binary operators.
- **Only existing operators can be overloaded**—you cannot create new ones.
- Related operators, like + and +=, **must be overloaded separately**.

Rules and Restrictions on Operator Overloading

As we prepare to overload operators for our own classes, there are several rules and restrictions you should keep in mind:

- **You cannot overload operators to change how an operator works on fundamental type values**. Operator overloading works only with objects of user-defined types or with a mixture of an object of a user-defined type and an object of a fundamental type:

  ```
  int operator+(int a, int b)
  { … }
  ```

  Error: 'int operator+(int, int)' **must have an argument of class or enumerated type**.

- When overloading (), [], -> or any of the assignment operators, the operator overloading function **must be declared as a class member**. For all other overloadable operators, the operator overloading functions can be member functions or non-member functions.

❖ Overload operators for class types so **they work as closely as possible to the way built-in operators work on fundamental types**.

Overloading Binary Operators

A binary operator can be overloaded

- as a non-static member function with **one** parameter or
- as a non-member function with **two** parameters (*one of those parameters must be either a class object or a reference to a class object*). A non-member operator function often is declared as friend of a class for performance reasons.

Binary Overloaded Operators as Member Functions

If y and z are String-class objects, then y < z is treated by the compiler as if y.operator<(z) had been written, invoking the operator< member function with one argument declared below:

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

➢ Overloaded operator functions for binary operators can be member functions ***only*** when the ***left*** operand is an object of the class in which the function is a member.

Overloading Binary Operators

Binary Overloaded Operators as Non-Member Functions

- As a non-member function, binary operator< *must* take *two* arguments—*one* of which *must* be an object (or a reference to an object) of the class that the overloaded operator is associated with.

- If y and z are String-class objects or references to String-class objects, then y < z is treated as if the call operator<(y, z) had been written in the program, invoking function operator<, which is declared as follows:

  bool **operator<**(const String&, const String&);

Overloading << and >> operators

- You can input and output fundamental-type data using the **stream extraction operator >>** and the **stream insertion operator <<**, respectively.

- The C++ class libraries overload these binary operators for each fundamental type, including pointers and char * strings.

- You can also overload these operators to perform input and output for your own types.

Overloading << and >> operators (example)

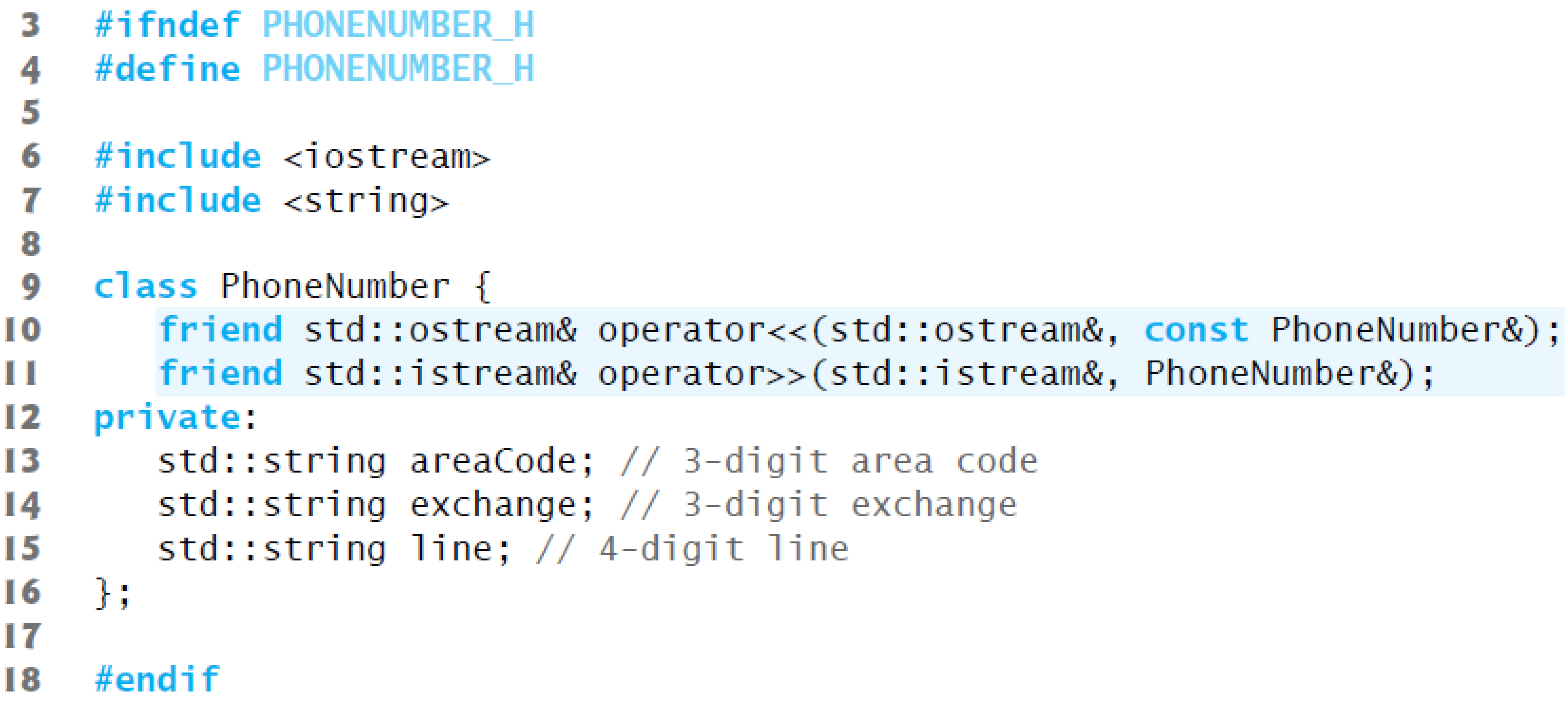
The program overloads these operators to input **PhoneNumber** objects in the format

```
(555) 555-5555
```

and to output them in the format

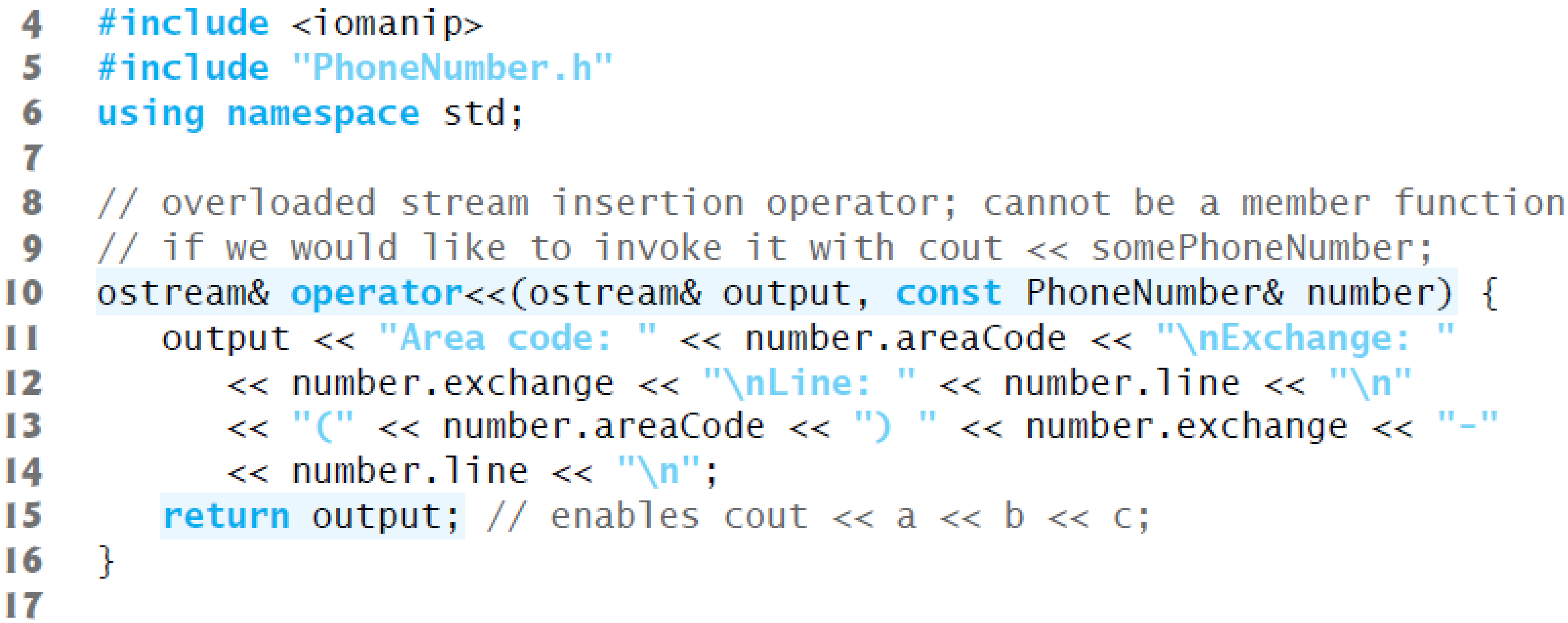
```
Area code: 555
Exchange: 555
Line: 5555
(555) 555-5555
```

Overloading << and >> operators (example)



```cpp
#ifndef PHONENUMBER_H
#define PHONENUMBER_H

#include <iostream>
#include <string>

class PhoneNumber {
   friend std::ostream& operator<<(std::ostream&, const PhoneNumber&);
   friend std::istream& operator>>(std::istream&, PhoneNumber&);
private:
   std::string areaCode; // 3-digit area code
   std::string exchange; // 3-digit exchange
   std::string line; // 4-digit line
};

#endif
```

Overloading << and >> operators (example)



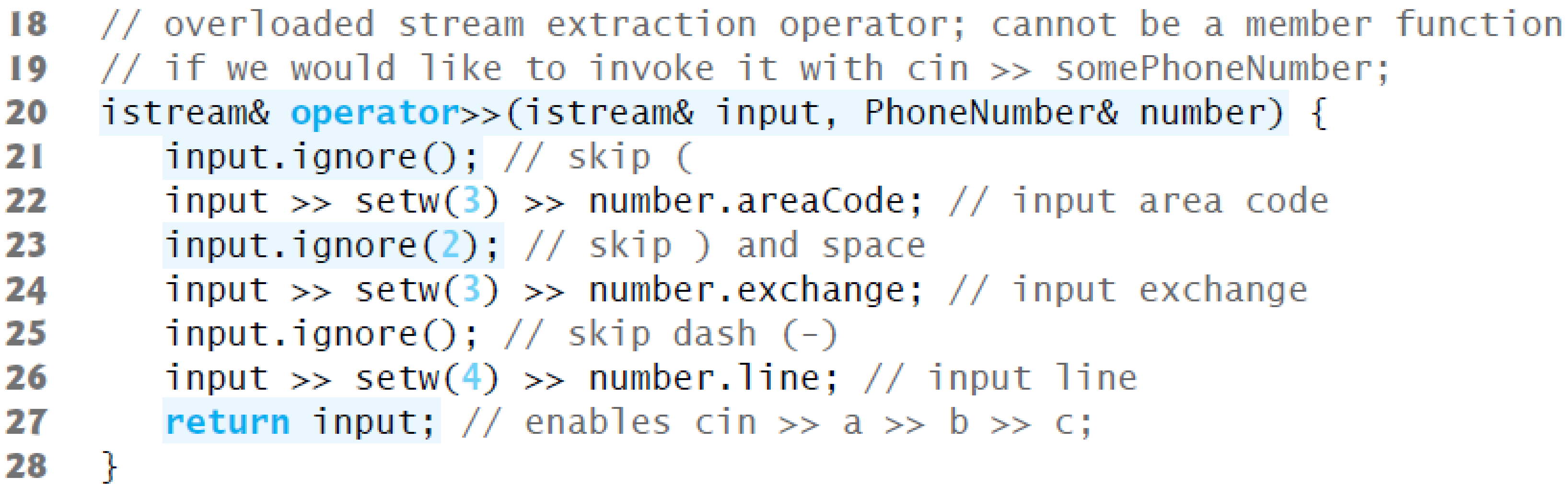
```
#include <iomanip>
#include "PhoneNumber.h"
using namespace std;

// overloaded stream insertion operator; cannot be a member function
// if we would like to invoke it with cout << somePhoneNumber;
ostream& operator<<(ostream& output, const PhoneNumber& number) {
   output << "Area code: " << number.areaCode << "\nExchange: "
      << number.exchange << "\nLine: " << number.line << "\n"
      << "(" << number.areaCode << ") " << number.exchange << "-"
      << number.line << "\n";
   return output; // enables cout << a << b << c;
}
```

- When the compiler sees the expression **cout << phone,** generates the non-member function call **operator<<(cout, phone);**
- The stream insertion operator function takes an ostream reference (output) and a const PhoneNumber reference (number) as arguments and returns an ostream reference to **enable cascaded calls:**

  cout << phone1 << phone2;

First, operator<<(cout, phone1) non-member function is called, which returns **ostream&.** Then the result is used to call **cout << phone2**

Overloading << and >> operators (example)



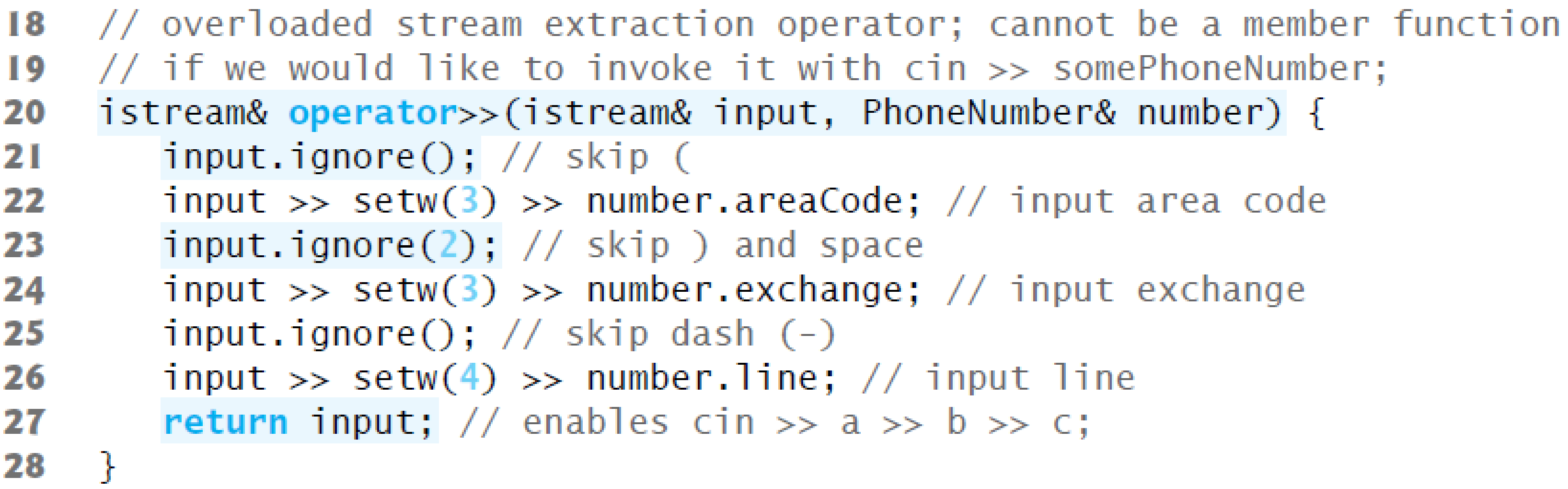
```
// overloaded stream extraction operator; cannot be a member function
// if we would like to invoke it with cin >> somePhoneNumber;
istream& operator>>(istream& input, PhoneNumber& number) {
   input.ignore(); // skip (
   input >> setw(3) >> number.areaCode; // input area code
   input.ignore(2); // skip ) and space
   input >> setw(3) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw(4) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
}
```

- When the compiler sees the expression **cin >> phone,** generates the non-member function call **operator>>(cin, phone);**
- The stream insertion operator function takes an istream reference (input) and a PhoneNumber reference (number) as arguments and returns an istream reference to **enable cascaded calls:**

  cin >> phone1 >> phone2;

First, operator>>(cin, phone1) non-member function is called, which returns **istream&.** Then the result is used to call **cin >> phone2**

Overloading << and >> operators (example)



```
// overloaded stream extraction operator; cannot be a member function
// if we would like to invoke it with cin >> somePhoneNumber;
istream& operator>>(istream& input, PhoneNumber& number) {
   input.ignore(); // skip (
   input >> setw(3) >> number.areaCode; // input area code
   input.ignore(2); // skip ) and space
   input >> setw(3) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw(4) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
}
```

- **operator>>** is declared as a friend of the PhoneNumber class so it can access a PhoneNumber's private members.
- When used with cin and strings, setw **restricts the number of characters read** to the number of characters specified by its argument (i.e., setw(3) allows three characters to be read).
- The parentheses, space and dash characters are skipped by calling istream member function **ignore.**

❖ Overloaded operators should mimic the functionality of their built-in counterparts—e.g., the + operator should perform addition, not subtraction. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

```cpp
#include <iostream>
#include "PhoneNumber.h"
using namespace std;

int main() {
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (555) 555-5555:" << endl;

   // cin >> phone invokes operator>> by implicitly issuing
   // the non-member function call operator>>(cin, phone)
   cin >> phone;

   cout << "\nThe phone number entered was:\n";

   // cout << phone invokes operator<< by implicitly issuing
   // the non-member function call operator<<(cout, phone)
   cout << phone << endl;
}
```

```
Enter phone number in the form (555) 555-5555:
(800) 555-1212

The phone number entered was:
Area code: 800
Exchange: 555
Line: 1212
(800) 555-1212
```

Overloading << and >> operators

Overloaded Operators as Non-Member friend Functions
The operator>> and operator<< are declared in PhoneNumber as *non-member, friend functions* **because the object of class PhoneNumber must be the operator's *right* operand.**

If these were to be PhoneNumber *member functions*, the following **confusing** statements would have to be used to output and input a PhoneNumber, respectively

```
phone << cout;
phone >> cin;
```

- New input/output capabilities for user-defined types are added to C++ without modifying standard input/output library classes. This is another example of C++'s **extensibility**.

- The only way to declare the operator<< as member function which will enable "cout<<phone" syntax is to declare it within ostream class. **This is not possible for user-defined classes**, since we are not allowed to modify C++ Standard Library classes.

Overloading Unary operators

A **unary operator** for a class can be overloaded as

- a non-static member function with no arguments or
- as a non-member function with one argument that must be an object (or a reference to an object) of the class.

Member functions that implement overloaded operators **must be non-static** so that they can access the non-static data in each object of the class.

Overloading Unary operators

Unary Overloaded Operators as Member Functions

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

When a unary operator such as ! Is overloaded as a member function with no arguments and the compiler sees the expression

```
!s
```

(in which s is an object of class String), the compiler generates the function call

```
s.operator!()
```

Overloading Unary operators

Unary Overloaded Operators as Non-Member Functions

```
bool operator!(const String&);
```

A unary operator such as ! may be overloaded as a *non-member function* with one parameter.

In this case, when the compiler sees the expression

```
!s
```

(in which s is an object of class String), the compiler generates the function call

```
operator!(s)
```

Overloading Increment/Decrement operators

The prefix and postfix versions of the increment and decrement operators can all be overloaded.

To overload the prefix and postfix increment operators, each overloaded operator function must have a **distinct signature**, so that the compiler will be able to determine which version of ++ is intended.

The **prefix** versions are overloaded exactly as any other prefix unary operator would be.

Overloading Increment/Decrement operators

Suppose that we want to add 1 to the day in a Date object named d1.

Overloading the Prefix Increment Operator

When the compiler sees the preincrementing expression **++d1**, if the overloaded operator is defined as a member function, the compiler generates the *member-function call*

d1.**operator**++()

The prototype for this operator member function would be

Date& **operator**++();

If the prefix increment operator is implemented as a *non-member function*, then, when the compiler sees the expression **++d1**, it generates the function call

operator++(d1)

The prototype for this non-member operator function would be declared as

Date& **operator**++(Date&);

Overloading Increment/Decrement operators

Overloading the Postfix Increment Operator

Overloading the postfix increment operator **presents a challenge**, because the compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions.

The convention that has been adopted is that, when the compiler sees the postincrementing expression d1++, it generates the member-function call

d1.**operator**++(**0**)

The prototype for this operator member function would be

Date **operator**++(int);

- The argument 0 is strictly a *dummy value* that enables the compiler to distinguish between the prefix and postfix increment operator functions.

- The same syntax is used to differentiate between the prefix and postfix **decrement** operator functions.

Overloading Increment/Decrement operators

Overloading the <u>Postfix Increment Operator</u>

If the postfix increment is implemented as a *non-member function*, then, when the compiler sees the expression d1++, the compiler generates the function call

operator++(d1, **0**)

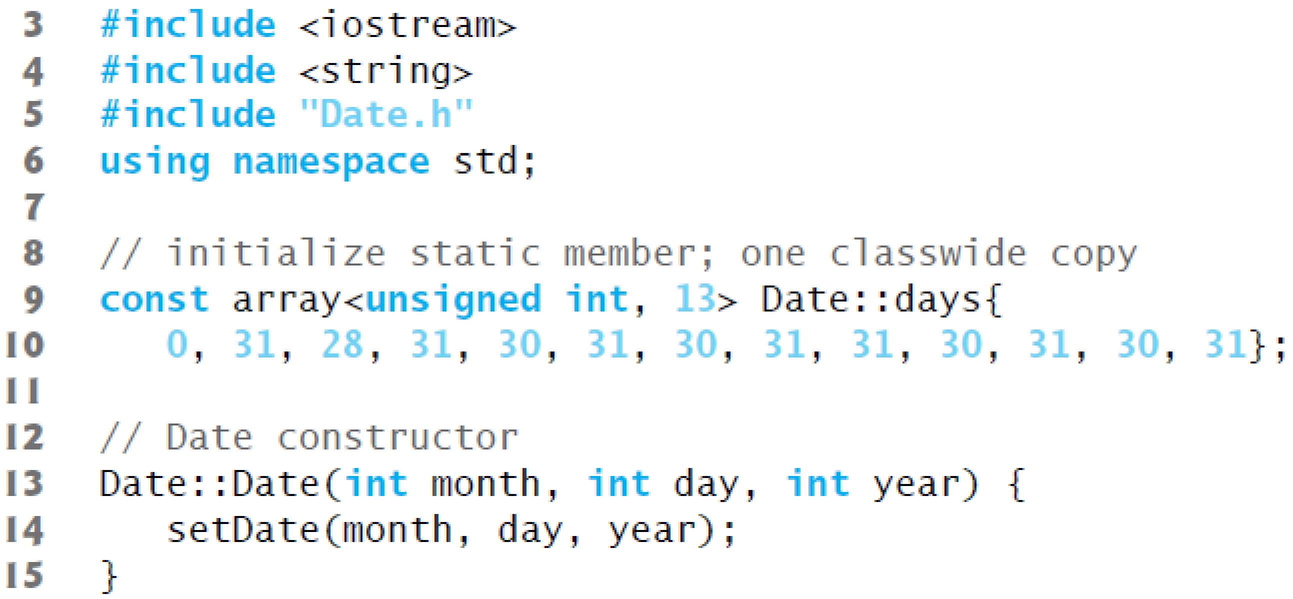
The prototype for this function would be

Date **operator**++(Date&, **int**);

- Note that the *postfix increment operator* returns Date objects ***by value***, whereas the prefix increment operator returns Date objects ***by reference****.*
- The postfix increment operator typically returns a <u>**temporary object**</u> **that contains the original value of the object before the increment occurred**.

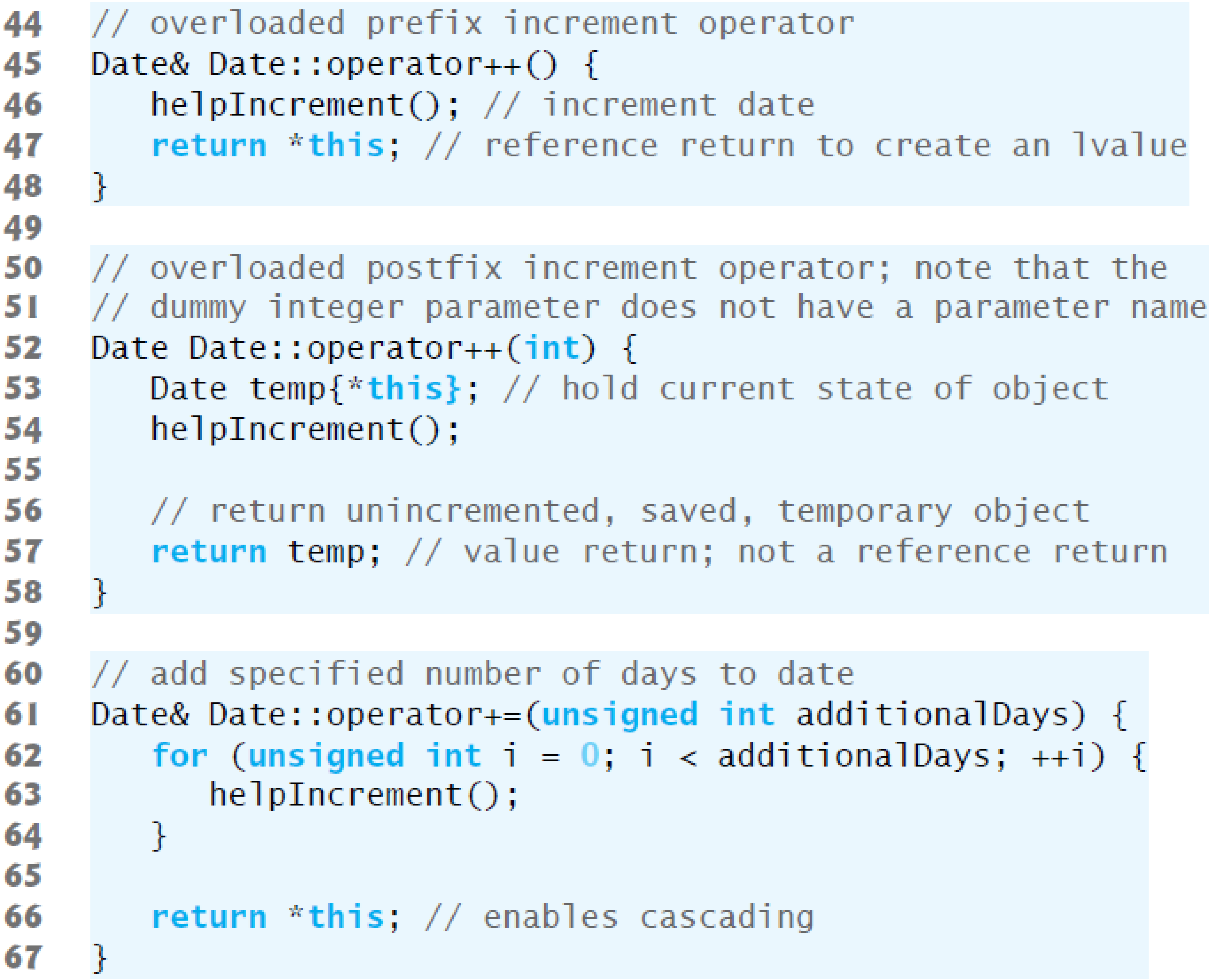
❖ The <u>extra object</u> that's created by the postfix increment (or decrement) operator can result in a <u>performance problem</u>—especially when the operator is used in a **loop**. For this reason, <u>you should prefer the overloaded **prefix** increment and decrement operators</u>.

Overloading Increment/Decrement operators

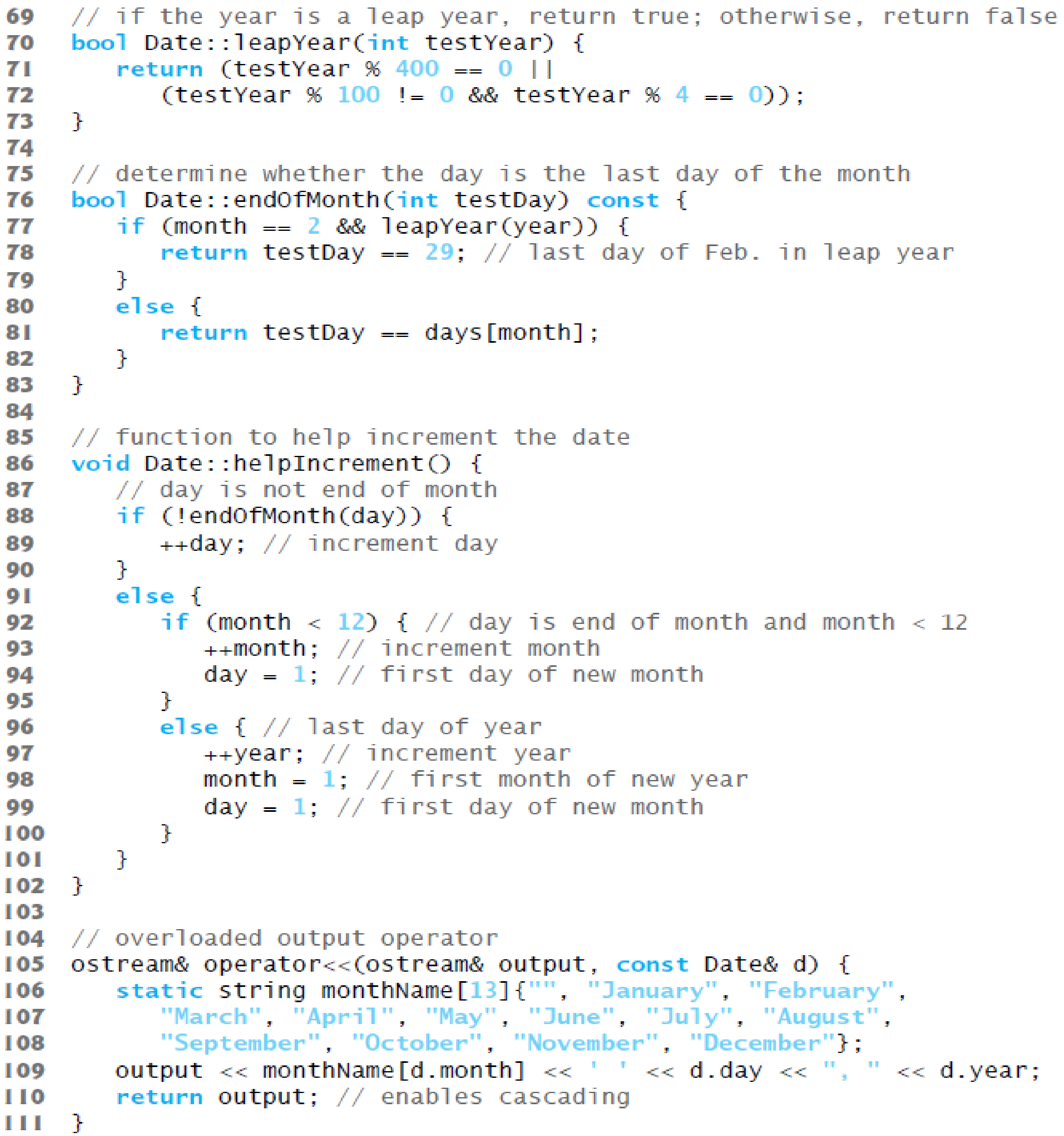


```cpp
#include <iostream>
#include <string>
#include "Date.h"
using namespace std;

// initialize static member; one classwide copy
const array<unsigned int, 13> Date::days{
   0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

// Date constructor
Date::Date(int month, int day, int year) {
   setDate(month, day, year);
}
```

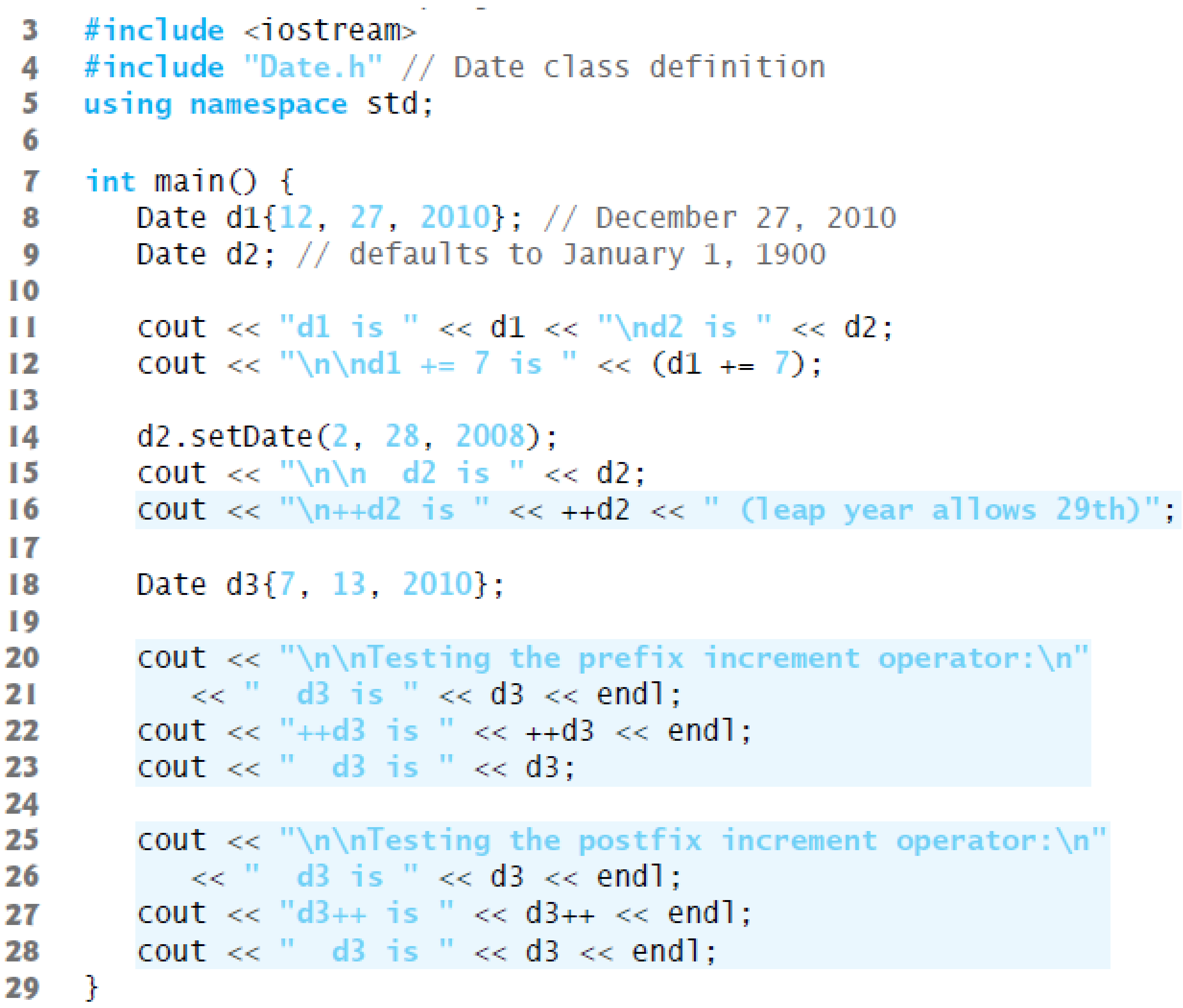
```cpp

// set month, day and year
void Date::setDate(int mm, int dd, int yy) {
   if (mm >= 1 && mm <= 12) {
      month = mm;
   }
   else {
      throw invalid_argument{"Month must be 1-12"};
   }

   if (yy >= 1900 && yy <= 2100) {
      year = yy;
   }
   else {
      throw invalid_argument{"Year must be >= 1900 and <= 2100"};
   }

   // test for a leap year
   if ((mn == 2 && leapYear(year) && dd >= 1 && dd <= 29) ||
        (dd >= 1 && dd <= days[mn])) {
      day = dd;
   }
   else {
      throw invalid_argument{
         "Day is out of range for current month and year"};
   }
}
```



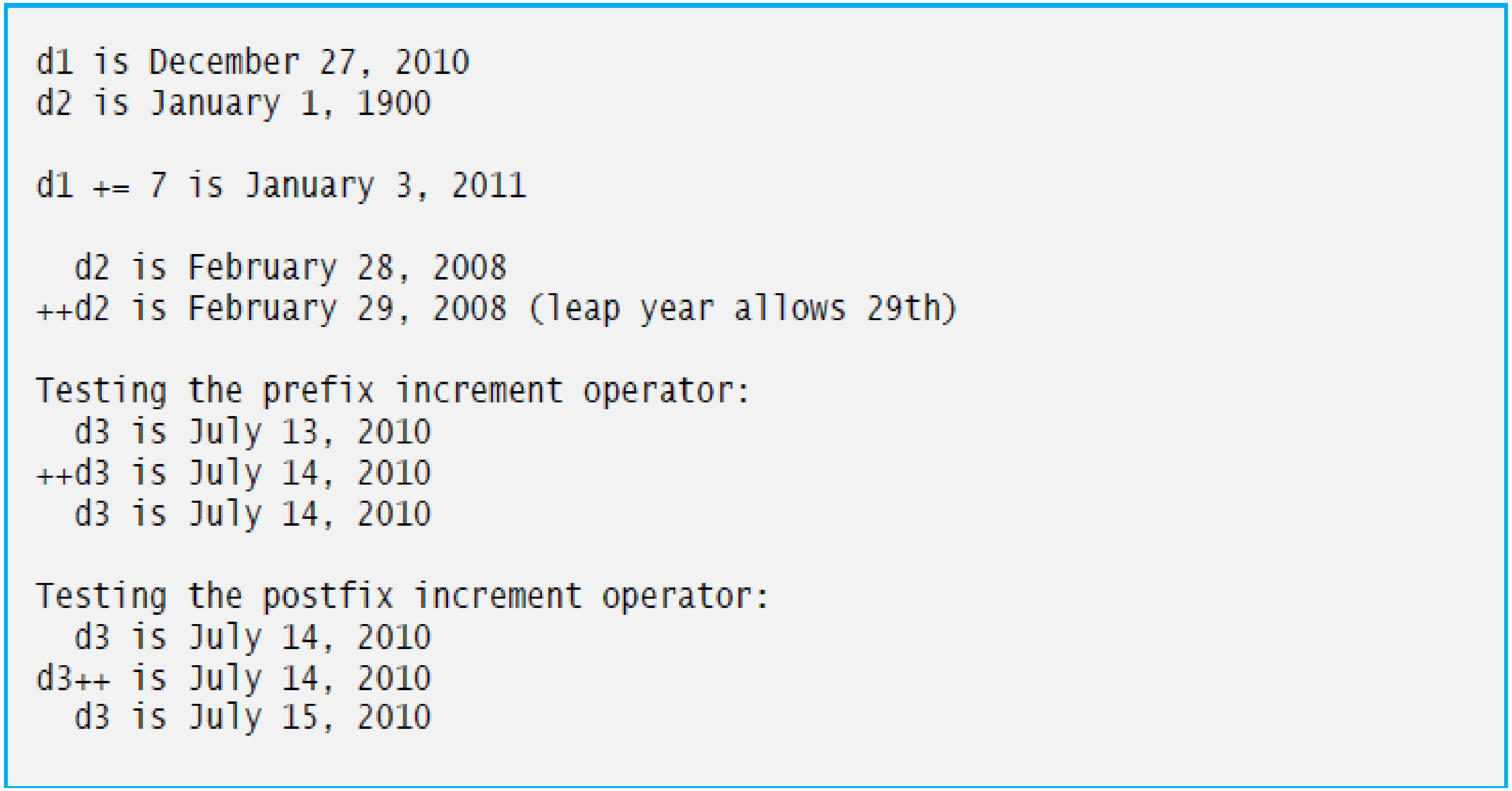
```cpp
// overloaded prefix increment operator
Date& Date::operator++() {
   helpIncrement(); // increment date
   return *this; // reference return to create an lvalue
}

// overloaded postfix increment operator; note that the
// dummy integer parameter does not have a parameter name
Date Date::operator++(int) {
   Date temp{*this}; // hold current state of object
   helpIncrement();

   // return unincremented, saved, temporary object
   return temp; // value return; not a reference return
}

// add specified number of days to date
Date& Date::operator+=(unsigned int additionalDays) {
   for (unsigned int i = 0; i < additionalDays; ++i) {
      helpIncrement();
   }

   return *this; // enables cascading
}
```



```cpp
// if the year is a leap year, return true; otherwise, return false
bool Date::leapYear(int testYear) {
   return (testYear % 400 == 0 ||
      (testYear % 100 != 0 && testYear % 4 == 0));
}

// determine whether the day is the last day of the month
bool Date::endOfMonth(int testDay) const {
   if (month == 2 && leapYear(year)) {
      return testDay == 29; // last day of Feb. in leap year
   }
   else {
      return testDay == days[month];
   }
}

// function to help increment the date
void Date::helpIncrement() {
   // day is not end of month
   if (!endOfMonth(day)) {
      ++day; // increment day
   }
   else {
      if (month < 12) { // day is end of month and month < 12
         ++month; // increment month
         day = 1; // first day of new month
      }
      else { // last day of year
         ++year; // increment year
         month = 1; // first month of new year
         day = 1; // first day of new month
      }
   }
}

// overloaded output operator
ostream& operator<<(ostream& output, const Date& d) {
   static string monthName[13]{"", "January", "February",
      "March", "April", "May", "June", "July", "August",
      "September", "October", "November", "December"};
   output << monthName[d.month] << ' ' << d.day << ", " << d.year;
   return output; // enables cascading
}
```



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

int main() {
   Date d1{12, 27, 2010}; // December 27, 2010
   Date d2; // defaults to January 1, 1900

   cout << "d1 is " << d1 << "\nd2 is " << d2;
   cout << "\n\nd1 += 7 is " << (d1 += 7);

   d2.setDate(2, 28, 2008);
   cout << "\n\n  d2 is " << d2;
   cout << "\n++d2 is " << ++d2 << " (leap year allows 29th)";

   Date d3{7, 13, 2010};

   cout << "\n\nTesting the prefix increment operator:\n"
      << "  d3 is " << d3 << endl;
   cout << "++d3 is " << ++d3 << endl;
   cout << "  d3 is " << d3;

   cout << "\n\nTesting the postfix increment operator:\n"
      << "  d3 is " << d3 << endl;
   cout << "d3++ is " << d3++ << endl;
   cout << "  d3 is " << d3 << endl;
}
```



```
d1 is December 27, 2010
d2 is January 1, 1900

d1 += 7 is January 3, 2011

  d2 is February 28, 2008
++d2 is February 29, 2008 (leap year allows 29th)

Testing the prefix increment operator:
  d3 is July 13, 2010
++d3 is July 14, 2010
  d3 is July 14, 2010

Testing the postfix increment operator:
  d3 is July 14, 2010
d3++ is July 14, 2010
  d3 is July 15, 2010
```