Lesson 6 - Operator Overloading

Outline

1. Introduction
2. Restrictions on Operator Overloading
3. Operator Functions as Class Members vs. as friend Functions
4. Overloading Unary Operators
5. Overloading Binary Operators
6. Overloading Stream-Insertion and Stream-Extraction Operators
7. Overloading ++ and - -
8. Case Study: A `Date` Class

Introduction

- Operator overloading
  - Enabling C++’s operators to work with class objects
  - Requires great care => programs difficult to understand
  - Compiler generates the appropriate code
- Overloading an operator
  - Function name is keyword **`operator`** followed by the symbol for the operator being overloaded
  - **`operator+`** used to overload the addition operator (**`+`**)
- Using operators
  - *To use an operator it must be overloaded*, but
    - the assignment operator **`(=)`** and the address operator **`(&)`** have default behavior, and may not be overloaded

Operator Overloading

- C++ operators that can be overloaded

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

- C++ Operators that cannot be overloaded

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

Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence of an operator cannot be changed
  - Associativity of an operator cannot be changed
  - Arity (number of operands) cannot be changed
    - Unary operators remain unary, and binary operators remain binary
    - Operators **&**, *****, **+** and **-** each have unary and binary versions
    - Unary and binary versions can be overloaded separately
- No new operators can be created
  - Use only existing operators
- No overloading operators for built-in types
  - Cannot change how two integers are added
  - Produces a syntax error

Class Members vs friend Functions

- Member vs non-member
  - Operator functions can be member or non-member functions
  - When overloading **`( )`**, **`[ ]`**, **`->`** or any of the assignment operators, *must use a member function*
- Operator functions as member functions
  - Leftmost operand must be an object of the class (or reference)
- Operator functions as non-member functions
  - *If left operand of a different type, operator function must be a non-member function*
  - Must be **`friend`**s if needs to access private or protected members
  - Enable the operator to be commutative

Overloading Unary Operators

- Unary operators
  - Can be overloaded with no arguments or one argument
  - **Should usually be implemented as member functions**
    - Avoid `friend` functions => violate the encapsulation
  - Example declaration as a member function:

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

  - Example declaration as a non-member function (**to be avoided**)

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

Overloading Binary Operators

- Binary operator as a member function:
  - *Non-static member function, one argument*

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

    - `y += z` is equivalent to `y.operator+=( z )`
- Binary operator as a non-member function:
  - *Non-member function, two arguments*

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

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

Stream-Insertion and Stream-Extraction

- A special case of binary operators:
  - **`operator<<`** and **`operator>>`**
- Overloaded to perform input/output for user-defined types
  - Left operand of types **`ostream &`** and **`istream &`**
  - *Must be a non-member function*
    - because left operand is not an object of the class
  - *Must be a* ***`friend`*** *function*
    - to access private data members

Outline

1. Class definition

1.1 Function definitions

```
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

class PhoneNumber {
   friend ostream &operator<<( ostream&, const PhoneNumber & );
   friend istream &operator>>( istream&, PhoneNumber & );

private:
   char areaCode[ 4 ];  // 3-digit area code and null
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null
};

// 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 &num )
{
```

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

They must be **`friend`** functions.

Outline

1.1 Function definition

1.2 Initialize variables

2. Get input

2.1 Assign to object

```
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;
   return output;     // enables cout << a << b << c;
}

istream &operator>>( istream &input, PhoneNumber &num )
{
   input.ignore();                     // skip (
   input >> setw( 4 ) >> num.areaCode; // input area code
   input.ignore( 2 );                  // skip ) and space
   input >> setw( 4 ) >> num.exchange; // input exchange
   input.ignore();                     // skip dash
   input >> setw( 5 ) >> num.line;     // input li
   return input;      // enables cin >> a >> b >> c
}

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

   cout << "Enter phone number in the form (123) 456-7890:\n";

   // cin >> phone invokes operator>> function by
   // issuing the call operator>>( cin, phone ).
   cin >> phone;

   // cout << phone invokes operator<< function by
   // issuing the call operator<<( cout, phone ).
   cout << "The phone number entered was: " << phone << endl;
   return 0;
}
```

The function call

`cin >> phone;`

interpreted as

`operator>>(cin, phone);`

`input` is an alias for **`cin`**, and **`num`** is an alias for **`phone`**.

Overloading ++ and - -

- Pre/post incrementing/decrementing operators
  - Allowed to be overloaded
  - Distinguishing between pre and post operators
    - prefix version:

```
d1.operator++();        // for ++d1
```

    - Convention for postincrementing expression:

```
d1.operator++( 0 );   // for d1++
```

    - `0` is a dummy value to make the argument list of `operator++` distinguishable from the argument list for `++operator`

Date Class

- Overloading operator<<
- Overloading operator+=
- Overloading unary operator ++
  - Preincrement
  - Post increment

```
// Fig. 8.6: date1.h
// Definition of class Date
#ifndef DATE1_H
#define DATE1_H
#include <iostream>

using std::ostream;

class Date {
   friend ostream &operator<<( ostream &, const Date & );

public:
   Date( int m = 1, int d = 1, int y = 1900 ); // constructor
   void setDate( int, int, int ); // set the date
   Date &operator++();            // preincrement operator
   Date operator++( int );        // postincrement operator
   const Date &operator+=( int ); // add days, modify object
   bool leapYear( int ) const;    // is this a leap year?
   bool endOfMonth( int ) const;  // is this end of month?

private:
   int month;
   int day;
   int year;

   static const int days[];       // array of days per month
   void helpIncrement();          // utility function
};

#endif
```

Outline

1. Class definition

1.1 Member functions

1.2 Member variables

```cpp
// Fig. 8.6: date1.cpp
// Member function definitions for Date class
#include <iostream>
#include "date1.h"

// Initialize static member at file scope;
// one class-wide copy.
const int Date::days[] = { 0, 31, 28, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31 };

// Date constructor
Date::Date( int m, int d, int y ) { setDate( m, d, y ); }

// Set the date
void Date::setDate( int mm, int dd, int yy )
{
   month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
   year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;

   // test for a leap year
   if ( month == 2 && leapYear( year ) )
      day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
   else
      day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
}

// Preincrement operator overloaded as a member function.
Date &Date::operator++()
{
   helpIncrement();
   return *this;  // reference return to create an lvalue
}

```

Outline

1. Load header

1.1 Define `days[]`

1.2 Function definitions

1.3 Constructor

1.4 `operator++` (preincrement)

```
// Postincrement operator overloaded as a member function.
// Note that the dummy integer parameter does not have a
// parameter name.
Date Date::operator++( int )
{
   Date temp = *this;
   helpIncrement();

   // return non-incremented, saved, temporary object
   return temp;   // value return; not a reference return
}

// Add a specific number of days to a date
const Date &Date::operator+=( int additionalDays )
{
   for ( int i = 0; i < additionalDays; i++ )
      helpIncrement();

   return *this;    // enables cascading
}

// If the year is a leap year, return true;
// otherwise, return false
bool Date::leapYear( int y ) const
{
   if ( y % 400 == 0 || ( y % 100 != 0 && y % 4 == 0 ) )
      return true;   // a leap year
   else
      return false;  // not a leap year
}

// Determine if the day is the end of the month
bool Date::endOfMonth( int d ) const
{
```

postincrement operator has a dummy `int` value.

Outline

1.5 `operator++(int)` (postincrement)

1.6 `operator+=`

1.7 `leapYear`

1.8 `endOfMonth`

```
   if ( month == 2 && leapYear( year ) )
      return d == 29; // last day of Feb. in leap year
   else
      return d == days[ month ];
}

// Function to help increment the date
void Date::helpIncrement()
{
   if ( endOfMonth( day ) && month == 12 ) {  // end year
      day = 1;
      month = 1;
      ++year;
   }
   else if ( endOfMonth( day ) ) {            // end month
      day = 1;
      ++month;
   }
   else       // not end of month or year; increment day
      ++day;
}

// Overloaded output operator
ostream &operator<<( ostream &output, const Date &d )
{
   static char *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
}
```

Outline

1.9 `helpIncrement`

1.10 `operator<<`
(output `Date`)

Outline

1. Load header

ects

s

3. Print results

```
// Fig. 8.6: fig08_06.cpp
// Driver for class Date
#include <iostream>

using std::cout;
using std::endl;

#include "date1.h"

int main()
{
   Date d1, d2( 12, 27, 1992 ), d3( 0, 99, 8045 );
   cout << "d1 is " << d1
        << "\nd2 is " << d2
        << "\nd3 is " << d3 << "\n\n";

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

   d3.setDate( 2, 28, 1992 );
   cout << "  d3 is " << d3;
   cout << "\n++d3 is " << ++d3 << "\n\n";

   Date d4( 3, 18, 1969 );

   cout << "Testing the preincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "++d4 is " << ++d4 << '\n';
   cout << "  d4 is " << d4 << "\n\n";

   cout << "Testing the postincrement operator:\n"
        << "  d4 is " << d4 << '\n';
   cout << "d4++ is " << d4++ << '\n';
   cout << "  d4 is " << d4 << endl;

   return 0;
}
```

```
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 March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969
```

```
Testing the preincrement operator:
  d4 is March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969
```

Program Output

```
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 March 18, 1969
++d4 is March 19, 1969
  d4 is March 19, 1969

Testing the postincrement operator:
  d4 is March 19, 1969
d4++ is March 19, 1969
  d4 is March 20, 1969
```