Operator Overloading

Contents

What is Operator Overloading

- Operator overloading: how to enable C++’s operators to work with objects
- Example: << (Stream Insertion) (also, bit-wise left-shift)
- The C++ language overloads the addition operator (+) and the subtraction operator (-). These operators perform differently, depending on their context in integer, floating-point and pointer arithmetic.
- 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.
- Overloading is especially appropriate for mathematical classes.

Restrictions on Operator Overloading

<table>
<tr><th colspan="8">Operators that can be overloaded</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td><td>-=</td><td>*=</td></tr>
<tr><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td><<</td><td>>></td><td>>>=</td></tr>
<tr><td><<=</td><td>==</td><td>!=</td><td><=</td><td>>=</td><td>&&</td><td>||</td><td>++</td></tr>
<tr><td>--</td><td>->*</td><td>,</td><td>-></td><td>[]</td><td>()</td><td>new</td><td>delete</td></tr>
<tr><td>new[]</td><td>delete[]</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><th colspan="4">Operators that cannot be overloaded</th></tr>
<tr><td>.</td><td>.*</td><td>::</td><td>?:</td></tr>
</table>

Overloading Addition Operator

```cpp
#ifndef TEST_H
#define TEST_H


class Test
{
    public:

        int num;
        Test();
        Test(int);
        Test operator+(Test);
};

#endif // TEST_H
```

```cpp
#include<iostream>
#include "Test.h"

Test::Test()
{

}

Test::Test(int a)
{
    num=a;
}


Test Test::operator+(Test obj)
{

    Test newObject;

    newObject.num=num+obj.num;

    return (newObject);

}
```

Overloading Addition Operator

```
#include<iostream>
#include"Test.h"

using namespace std;

main()
{
    Test obj1(100);
    Test obj2(200);


    cout << "obj1.num=" << obj1.num << endl;
    cout << "obj2.num=" << obj2.num << endl;


    Test obj3;
    cout << "Blank obj3.num=" << obj3.num << endl;


    obj3=obj1+obj2;

    cout << "Overlaoding obj3.num=" << obj3.num << endl;
}
```

```
obj1.num=100
obj2.num=200
Blank obj3.num=4310016
Overlaoding obj3.num=300

Process returned 0 (0x0)   execution time : 0.017 s
Press any key to continue.
```

Overloading Stream Insertion and Extraction Operators

- We can Input and output fundamental-type data using the stream extraction operator >> and the stream insertion operator <<.

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

- Also overload these operators to perform input and output for our own types.

- Overloads these operators to input and output PhoneNumber objects in the format “(000) 000-0000.”

PhoneNumber class with overloaded stream insertion and stream extraction operators as friend functions

```
// Fig. 11.3: PhoneNumber.h
// PhoneNumber class definition
#ifndef PHONENUMBER_H
#define PHONENUMBER_H

#include <iostream>
#include <string>
using namespace std;

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

#endif
```

(000) 000-0000

Overloaded stream insertion and stream extraction operators for class PhoneNumber.

```
// Fig. 11.4: PhoneNumber.cpp

#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 << "(" << number.areaCode << ") "
      << number.exchange << "-" << number.line;
   return output; // enables cout << a << b << c;
} // end function operator<<

// 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;
} // end function operator>>
```

Overloaded stream insertion and stream extraction operators

```cpp
// Fig. 11.5: fig11_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
#include "PhoneNumber.h"
using namespace std;

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

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

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

   cout << "The phone number entered was: ";

   // cout << phone invokes operator<< by implicitly issuing
   // the global function call operator<<( cout, phone )
   cout << phone << endl;
} // end main
```

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

Dynamic Memory Management

- A standard C++ **array data structure is fixed in size** once it's created. The size is specified with a constant at compile time.

- Sometimes it's useful to determine the size of an array dynamically at execution time and then create the array.

- C++ enables us to **control the allocation and deallocation of memory** in a program *for objects and for arrays* of any built-in or user-defined type.

- This is known as dynamic memory management and is performed with the operators **new** and delete.

Dynamic Memory Management

- Obtaining Dynamic Memory

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

The new operator allocates storage of the proper size for an object of type Time, calls the default constructor to initialize the object and returns a pointer to the type specified to the right of the new operator (i.e., a Time *)

- Releasing Dynamic Memory

```
delete timePtr;
```

This statement first calls the destructor for the object to which timePtr points, then deallocates the memory associated with the object, returning the memory to the free store.

Dynamic Memory Management

- Initializing Dynamic Memory

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

```
Time *timePtr = new Time( 12, 45, 0 );
```

- Dynamically Allocating Arrays

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

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

- Releasing dynamically allocating Arrays

```
delete [] gradesArray;
```

Proxy Classes

- A proxy class allows us to hide even the private data of a class from clients of the class.

- Providing clients of a class with **a proxy class that knows only the public interface** to the class enables the clients to use the class's services without giving the clients access to the class's implementation details.

```
// Fig. 11.16: Implementation.h
// Implementation class definition.

class Implementation
{
public:
   // constructor
   Implementation( int v )
      : value( v ) // initialize value with v
   {
      // empty body
   } // end constructor Implementation

   // set value to v
   void setValue( int v )
   {
      value = v; // should validate v
   } // end function setValue

   // return value
   int getValue() const
   {
      return value;
   } // end function getValue
private:
   int value; // data that we would like to hide from the client
}; // end class Implementation
```

Proxy Class

```
// Fig. 11.17: Interface.h
// Proxy class Interface definition.
// Client sees this source code, but the source code does not reveal
// the data layout of class Implementation.

class Implementation; // forward class declaration required by line 17

class Interface
{
public:
   Interface( int ); // constructor
   void setValue( int ); // same public interface as
   int getValue() const; // class Implementation has
   ~Interface(); // destructor
private:
   // requires previous forward declaration (line 6)
   Implementation *ptr;
}; // end class Interface
```

Proxy Class

```
// Fig. 11.18: Interface.cpp
// Implementation of class Interface--client receives this file only
// as precompiled object code, keeping the implementation hidden.
#include "Interface.h" // Interface class definition
#include "Implementation.h" // Implementation class definition

// constructor
Interface::Interface( int v )
   : ptr ( new Implementation( v ) ) // initialize ptr to point to
{                                    // a new Implementation object
   // empty body
} // end Interface constructor

// call Implementation's setValue function
void Interface::setValue( int v )
{
   ptr->setValue( v );
} // end function setValue

// call Implementation's getValue function
int Interface::getValue() const
{
   return ptr->getValue();
} // end function getValue

// destructor
Interface::~Interface()
{
   delete ptr;
} // end ~Interface destructor
```

```cpp
// Fig. 11.19: fig11_19.cpp
// Hiding a class's private data with a proxy class.
#include <iostream>
#include "Interface.h" // Interface class definition
using namespace std;

int main()
{
   Interface i( 5 ); // create Interface object

   cout << "Interface contains: " << i.getValue()
      << " before setValue" << endl;

   i.setValue( 10 );

   cout << "Interface contains: " << i.getValue()
      << " after setValue" << endl;
} // end main
```

```
Interface contains: 5 before setValue
Interface contains: 10 after setValue
```