11.10 OVERLOADING THE SUBSCRIPT OPERATOR

Recall that, if `a` is an array, then the expression `a[i]` really means nothing more than `*(a+i)`. This is because `a` is actually the address of the initial element in the array, so `a+i` is the address of the *i*th element, since the number of bytes added to `a` is `i` times the size of each array element.

The symbol `[]` denotes the *subscript operator.* Its name derives from the original use of arrays, where `a[i]` represented the mathematical symbol a_i. When used as `a[i]`, it has two operands: `a` and `i`. The expression `a[i]` is equivalent to `operator[](a, i)`. And as an operator, `[]` can be overloaded.

EXAMPLE 11.13 Adding a Subscript Operator to the `Ratio` Class

```
#include <iostream>
#include <cstdlib>    // defines the exit() function

class Ratio
{   friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    int& operator[](int);
    // other declarations go here
  private:
    int num, den;
    // other declarations go here
};
int main()
{ Ratio x(22,7);
  cout << "x = " << x << endl;
  cout << "x[1] = " << x[1] << ", x[2] = " << x[2] << endl;
}
int& Ratio::operator[](int i)
{ if (i == 1) return num;
  if (i == 2) return den;
  cerr << "ERROR: index out of range\n";
  exit(0);
}
```

```
    x = 22/7
    x[1] = 22, x[2] = 7
```

The expression `x[1]` calls the subscript operator, passing 1 to i, which returns `x.num`. Similarly, `x[2]` returns `x.den`. If `i` has any value other than 1 or 2, then an error message is sent to `cerr`, the standard error stream, and then the `exit()` function is called.

This example is artificial. There is no advantage to accessing the fields of the `Ratio` object `x` with `x[1]` and `x[2]` instead of `x.num` and `x.den`. However, there are many important classes where the subscript is very useful. (See Problem 11.2.)

Note that the subscript operator is an access function, since it provides `public` access to `private` member data.

Review Questions

11.1 How is the `operator` keyword used?

11.2 What does `*this` always refer to?

11.3 Why can't the `this` pointer be used in nonmember functions?

11.4 Why should the overloaded assignment operator return `*this`?

11.5 What is the difference between the effects of the following two declarations:

```
Ratio y(x);
Ratio y = x;
```

11.6 What is the difference between the effects of the following two lines:

```
Ratio y = x;
Ratio y;  y = x;
```

11.7 Why can't `**` be overloaded as an exponentiation operator?

11.8 Why should the stream operators `<<` and `>>` be overloaded as `friend` functions?

11.9 Why should the arithmetic operators `+`, `-`, `*`, and `/` be overloaded as `friend` functions?

11.10 How is the overloaded pre-increment operator defintion distinguished from that of the overloaded post-increment operator?

11.11 Why is the `int` argument in the implementation of the post-increment operator left unnamed?

11.12 What mechanism allows the overloaded subscript operator `[]` to be used on the left side of an assignment statement, like this: `v[2] = 22`?

Problems

11.1 Implement the binary subtraction operator, the unary negation operator, and the less-than operator `<` for the `Ratio` class (see Example 11.4 on page 259).

11.2 Implement a `Vector` class, with a default constructor, a copy constructor, a destructor, and overloaded assignment operator, subscript operator, equality operator, stream insertion operator, and stream extraction operator.

11.3 Implement the addition and division operators for the `Ratio` class (see Example 11.5 on page 259).

11.4 Rewrite the overloaded input operator for the `Ratio` class (Example 11.9 on page 262) so that, instead of prompting for the numerator and denominator, it reads a fraction type as "`22/7`".

11.5 Implement an overloaded assignment operator `=` for the `Point` class (see Problem 10.1 on page 249).

11.6 Implement overloaded stream insertion operator `<<` for the `Point` class (see Problem 10.1 on page 249).

11.7 Implement overloaded comparison operators `==` and `!=` for the `Point` class (see Problem 10.1 on page 249).

11.8 Implement overloaded addition operator `+` and subtraction operator `-` for the `Point` class (see Problem 10.1 on page 249).

11.9 Implement an overloaded multiplication operator `*` to return the dot product of two `Point` objects (see Problem 10.1 on page 249).

Answers to Review Questions

11.1 The `operator` keyword is used to form the name of a function that overloads an operator. For example, the name of the function that overloads the assignment operator `=` is "`operator=`".

11.2 The keyword `this` is a pointer to the object that owns the call of the member function in which the expression appears.

11.3 The expression `*this` always refers to the object that owns the call of the member function in which the expression appears. Therefore, it can only be used within member functions.

11.4 The overloaded assignment operator should return `*this` so that the operator can be used in a cascade of calls, like this: `w = x = y = z;`

11.5 There is no difference. Both declarations use the copy constructor to create the object `y` as a duplicate of the object `x`.

11.6 The declaration `Ratio y = x;` calls the copy constructor. The code `Ratio y; y = x;` calls the default constructor and then the assignment operator.

11.7 The symbol `**` cannot be overloaded as an operator because it is not a C++ operator.

11.8 The stream operators `<<` and `>>` should be overloaded as `friend` functions because their left operands should be stream objects. If an overloaded operator is a member function, then its left operand is `*this`, which is an object of the class to which the function is a member.

11.9 The arithmetic operators `+`, `-`, `*`, and `/` should be overloaded as `friend` functions so that their left operands can be declared as `const`. This allows, for example, the use of an expression like `22 + x`. If an overloaded operator is a member function, then its left operand is `*this`, which is not `const`.

11.10 The overloaded pre-increment operator has no arguments. The overloaded post-increment operator has one (dummy) argument, of type `int`.

11.11 The `int` argument in the implementation of the post-increment operator is left unnamed because it is not used. It is a dummy argument.

11.12 By returning a reference, the overloaded subscript operator `[]` can be used on the left side of an assignment statement, like this: `v[2] = 22`. This is because, as a reference, `v[2]` is an lvalue.

Solutions to Problems

11.1 All three of these operators are implemented as `friend` functions to give them access to the `num` and `den` data members of their owner objects:

```
class Ratio
{   friend Ratio operator-(const Ratio&, const Ratio&);
    friend Ratio operator-(const Ratio&);
    friend bool operator<(const Ratio&, const Ratio&);
  public:
    Ratio(int =0, int =1);
    Ratio(const Ratio&);
    Ratio& operator=(const Ratio&);
    // other declarations go here
  private:
    int num, den;
    int gcd(int, int)
    int reduce();
};
```

The binary subtraction operator simply constructs and returns a `Ratio` object `z` that represents the difference `x - y`:

```
Ratio operator-(const Ratio& x, const Ratio& y)
{ Ratio z(x.num*y.den - y.num*x.den, x.den*y.den);
```

```
  z.reduce();
  return z;
}
```

Algebraically, the subtraction *a*/*b* - *c*/*d* is performed using the common denominator *bd*:

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

So the numerator of `x - y` should be `x.num*y.den - y.num*x.den` and the denominator should be `x.den*y.den`. The function constructs the `Ratio` object `z` with that numerator and denominator. This algebraic formula can produce a fraction that is not in reduced form, even if x and y are. For example, $1/2 - 1/6 = (1 \cdot 6 - 2 \cdot 1)/(2 \cdot 6) = 4/12$. So we call the `reduce()` utility function before returning the resulting object `z`.

The unary negation operator overloads the symbol "`-`". It is distinguished from the binary subtraction operator by its parameter list; it has only one parameter:

```
Ratio Ratio::operator-(const Ratio& x)
{ Ratio y(-x.num, x.den);
  return y;
}
```

To negate a fraction *a*/*b* we simply negate its numerator: (-*a*)/*b*. So the newly constructed `Ratio` object `y` has the same denominator as `x` but its numerator is `-x.num`. The less-than operator is easier to do if we first modify our default constructor to ensure that every object's `den` value is positive. Then we can use the standard equivalence for the less-than operator:

$$\frac{a}{b} < \frac{c}{d} \Leftrightarrow ad < bc$$

```
bool operator<(const Ratio& x, const Ratio& y)
{ return (x.num*y.den < y.num*x.den);
}

Ratio::Ratio(int n=0, int d=1) : num(n), den(d)
{ if (d == 0) n = 0;
  else if (d < 0) { n *= -1; d *= -1; }
  reduce();
}
```

The modification ensuring that `den > 0` could instead be done in the `reduce()` function, since that utility should be called by every member function that allows `den` to be changed. However, none of our other member functions allows the sign of `den` to change, so by requiring it to be positive when the object is constructed we don't need to check the condition again.

11.2 Here is the class declaration:

```
class Vector
{   friend bool operator==(const Vector&, const Vector&);
    friend ostream& operator<<(ostream&, const Vector&);
    friend istream& operator>>(istream&, Vector&);
  public:
    Vector(int =1, double =0.0);            // default constructor
    Vector(const Vector&);                  // copy constructor
    ~Vector();                              // destructor
    const Vector& operator=(const Vector&); // assignment operator
    double& operator[](int) const;          // subscript operator
  private:
    int size;
    double* data;
};
```

Here is the implementation of the overloaded equality operator:

```
bool operator==(const Vector& v, const Vector& w)
{ if (v.size != w.size) return 0;
  for (int i = 0; i < v.size; i++)
  if (v.data[i] != w.data[i]) return 0;
  return 1;
}
```

It is a nonmember function which returns 1 or 0 according to whether the two vectors `v` and `w` are equal. If their sizes are not equal, then it returns 0 immediately. Otherwise it checks the corresponding elements of the two vectors, one at a time. If there is any mismatch, then again it returns 0 immediately. Only if the entire loop finishes without finding any mismatches can we conclude that the two vectors are equal and return 1.

Here is the implementation of the overloaded stream insertion operator:

```
ostream& operator<<(ostream& ostr, const Vector& v)
{ ostr << '(';
  for (int i = 0; i < v.size-1; i++) {
  ostr << v[i] << ", ";
  if ((i+1)%8 == 0) cout << "\n ";
  }
  return ostr << v[i] << ")\n";
}
```

This prints the vector like this: `(1.11111, 2.22222, 3.33333, 4.44444, 5.55556)`. The conditional inside the loop allows the output to "wrap" around several lines neatly if the vector has more than 8 elements.

The output is sent to `ostr` which is just a local name for the output stream that is passed to the function. That would be `cout` if the function is called like this: `cout << v;`.

In the last line of the function, the expression `ostr << v[i] << ")\n"` makes two calls to the (standard) stream extraction operator. Those two calls return `ostr` as the value of this expression, and so that object `ostr` is then returned by this function.

Here is the overloaded stream extraction operator:

```
istream& operator>>(istream& istr, Vector& v)
{ for (int i = 0; i < v.size; i++)
  { cout << i << ": ";
    istr >> v[i];
  }
  return istr;
}
```

This implementation prompts the user for each element of the vector `v`. It could also be implemented without user prompts, simply reading the elements one at a time. Notice that the elements are read from the input stream `istr`, which is the first parameter passed in to the function. When the function is called like this: `cin >> v;` the standard input stream `cin` will be passed to the parameter `istr`, so the vector elements are actually read from `cin`. The argument `istr` is simply a local name for the actual input stream which probably will be `cin`. Notice that this argument is also returned, allowing a cascade of calls like this: `cin >> u >> v >> w;`.

Here is the implementation of the default constructor:

```
Vector::Vector(int sz=1, double t=0.0) : size(sz)
{ data = new double[size];
  for (int i = 0; i < size; i++)
    data[i] = t;
}
```

The declaration `Vector u;` would construct the vector `u` having 1 element with the value 0.0; the declaration `Vector v(4);` would construct the vector `v` with 4 elements all with the value 0.0; and the declaration `Vector w(8, 3.14159);` would construct the vector `w` with 8 elements all with the value 3.14159.

This constructor uses the initialization list `size(sz)` to assign the argument `sz` to the data member `size`. Then it uses the `new` operator to allocate that number of elements to the array `data`. Finally, it initializes each element with the value `t`.

The copy constructor is almost the same as the default constructor:

```
Vector::Vector(const Vector& v) : size(v.size)
{ data = new double[v.size];
  for (int i = 0; i < size; i++)
    data[i] = v.data[i];
}
```

It uses the data members of the vector argument `v` to initialize the object being constructed. So it assigns `v.size` to the new object's `size` member, and it assigns `v.data[i]` to the elements of the new object's `data` member.

The destructor simply restores the storage allocated to the `data` array and then sets `data` to `NULL` and `size` to 0:

```
Vector::~Vector()
{ delete [] data;
  data = NULL;
  size = 0;
}
```

The overloaded assignment operator creates a new object that duplicates the vector `v`:

```
const Vector& Vector::operator=(const Vector& v)
{ if (&v != this)
  { delete [] data;
    size = v.size;
    data = new double[v.size];
    for (int i = 0; i < size; i++)
      data[i] = v.data[i];
  }
  return *this;
}
```

The condition `(&v != this)` determines whether the object that owns the call is different from the vector `v`. If the address of `v` is the same as `this` (which is the address of the current object), then they are the same object and nothing needs to be done. This check is a safety precaution to guard against the possibility that an object might, directly or indirectly, be assigned to itself, like this: `w = v = w;`.

Before creating a new object, the function restores the allocated data array. Then it copies the vector `v` the same way that the copy constructor did.

The overloaded subscript operator simply returns the `i`th component of the object's `data` array:

```
double& Vector::operator[](int i) const
{ return data[i];
}
```

11.3

```
Ratio operator+(const Ratio& r1, const Ratio& r2)
{ Ratio r(r1.num*r2.den+r2.num*r1.den,r1.den*r2.den);
  r.reduce();
  return r;
}
Ratio operator/(const Ratio& r1, const Ratio& r2)
{ Ratio r(r1.num*r2.den,r1.den*r2.num);
```

```
  r.reduce();
  return r;
}
```

11.4

```
ostream& operator<<(ostream& ostr, const Ratio& r)
{ return ostr << r.num << "/" << r.den;
}
```

11.5

```
Point& Point::operator=(const Point& point)
{ _x = point._x;
  _y = point._y;
  _z = point._z;
  return *this;
}
```

11.6

```
ostream& operator<<(ostream& ostr, const Point& point)
{ return ostr << "(" << _x << "," << _y << "," << _z << ")";
}
```

11.7

```
bool Point::operator==(const Point& point) const
{ return _x == point._x && _y == point._y && _z == point._z;
}
bool Point::operator!=(const Point& point) const
{ return _x != point._x || _y != point._y || _z != point._z;
}
```

11.8

```
Point operator+(const Point& p1, const Point& p2)
{ return Point(p1._x+p2._x,p1._y+p2._y,p1._z+p2._z);
}
Point operator-(const Point& p1, const Point& p2)
{ return Point(p1._x-p2._x,p1._y-p2._y,p1._z-p2._z);
}
```

11.9

```
Point operator*(const double coef, const Point& point)
{ return Point(coef*point._x,coef*point._y,coef*point._z);
}
```