Overloaded operators for user-defined types

Michael R. Nowak

Texas A&M University

Basic concepts

- An overloaded operator is a function; it thus has a:
  - name (the keyword `operator` followed by the operator being defined)
  - return type
  - parameter list
  - body
- An overloaded operator has the same number of parameters as the operator has operands
  - That's a single parameter for a unary operator
  - Two parameters for a binary operator
    - The left-hand argument is used to initialize the first parameter and the right-hand argument the second

Basic concepts

- We can only overload existing operators; we cannot invent new operator symbols
- Four symbols (*, &, +, and -,) serve as both unary and binary operators
  - Either (or both) of these operators can be overloaded
  - It is the number of parameters that determines whether the unary or binary operator is being defined
- An overloaded operator will have the same precedence and associativity of that of a built-in operator
  - Therefore, regardless of the operand types, the expression `x = y + z` will evaluate to `x = (y + z)`

Basic concepts

- If the overloaded operator is defined as a member function, the first operand is bound to the *implicit* this pointer
  - Due to this, a member operator function will have one less (*explicit*) parameter than the operator has operands
- If the overloaded operator is defined as a non-member helper function, at least one of its parameters must be a user-defined type
  - This implies that we cannot change the meaning of an operator when applied to operands of built-in type

Calling an overloaded operator (non-member)

- We frequently use an overloaded operator indirectly, using the operator on arguments of a suitable type
  - `something + something_else`
- We can also call an overloaded operator using the function call syntax that you're familiar with
  - `operator+(something, something_else)`
- Both calling mechanisms are equivalent; both call the non-member function `operator+` passing `something` as the first arugment and `something_else` as the second

Calling an overloaded operator (member)

- We frequently use an overloaded operator indirectly, using the operator on arguments of a suitable type
  - `something + something_else`
- When we call an overloaded operator that's been declared as a member of our user-defined type using function call syntax, we
  - name an object on which to run the function,
  - use the dot operator to note the member function we are interested in calling,
  - and provide any necessary arguments
- For a binary `operator+` declared as a function member, we would write
  - `something.operator+(something_else)`

Member or nonmember implementation?

- We must decide whether to define an overloaded operator as a class member or a nonmember function
  - Some operators are required to be members; others may not be able to be correctly defined as such
- When overloading an operator, you should follow these guidelines:
  - Assignment (=), subscript(`[]`), call (`()`), and member access arrow (->) *must* be defined as *members*
  - Compound assignment operators (ex., +=) *should* be defined as *members*
  - Operators that change the state of an object *should* be defined as *members*
  - Symmetric operators *should* be defined as *non-members*
    - We want to use symmetric operators in expressions with mixed types
    - We couldn't do this if the operator is defined as a member; why?

Overloading operator=

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }
    Color& operator=(Color const& rhs);

private:
    int r; int g; int b;
};

#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r}, g{g}, b{b} {}

Color& Color::operator=(Color const& rhs)
{
    r = rhs.r; g = rhs.g; b = rhs.b;
    return *this;
}
```

Overloading operator+=

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }
    Color& operator+=(Color const& rhs);

private:
    int r; int g; int b;
};

#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r}, g{g}, b{b} {}

Color& Color::operator+=(Color const& rhs)
{
    r = (r + rhs.r) / 2;
    g = (g + rhs.g) / 2;
    b = (b + rhs.b) / 2;
    return *this;
}
```

Overloading `operator--` (prefix)

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }
    Color& operator--();

private:
    int r; int g; int b;
};

#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r}, g{g}, b{b} {}

Color& Color::operator--()
{
    // do something
    return *this;
}
```

Overloading `operator--` (postfix)

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }
    Color operator--(int);

private:
    int r; int g; int b;
};

#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r}, g{g}, b{b} {}

Color Color::operator--(int)
{
    Color old_state = *this; // why should we do this?
    // do something // what should this do?
    return old_state; // why should we return this?
}
```

Overloading operator+

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }

private:
    int r; int g; int b;
};

Color operator+(Color const& lhs, Color
const& rhs);

#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r}, g{g}, b{b} {}

Color operator+(Color const& lhs, Color const& rhs)
{
    return (Color{(lhs.get_r() +  rhs.get_r())/2,
                  (lhs.get_g() + rhs.get_g())/2,
                  (lhs.get_b() + rhs.get_b())/2});
}
```

Overloading operator<<

Color.h

```
#ifndef COLOR_H
#define COLOR_H
class Color {
public:
    Color();
    Color(int, int, int);
    int get_r() const { return r; }
    int get_g() const { return g; }
    int get_b() const { return b; }

private:
    int r; int g; int b;
};

std::ostream& operator<<(std::ostream &os,
Color const& rhs);

#endif
```

Color.cpp

```
#include "Color.h"

Color::Color() : r{0}, g{0}, b{0} {}
Color::Color(int r, int g, int b) : r{r}, g{g}, b{b} {}

std::ostream& operator<<(std::ostream &os,
  Color const& rhs)
{
    os << rhs.get_r() << '\t'
       << rhs.get_b() << '\t'
       << rhs.get_b();
     return os;
}
```