

Chapter 5: Operator Overloading

Department of Computer Science and Engineering
Kathmandu University

Rajani Chulyadyo, PhD

Contents

- Introduction to operator overloading
 - Overloading unary operators
 - Overloading binary operators
 - Overloading binary operators using friend functions
 - Overloading operators using normal functions
 - Copy assignment operator
-

Operator overloading

Recall function overloading

```
int add(int a, int b) { return a + b; }  
int add(int a, int b, int c) { return a + b + c; }
```

In C++, operators are implemented as functions.

By using function overloading on the operator functions, we can define our own versions of the operators that work with different data types (including classes).

Operator overloading

An example

```
// The same operator + performs different tasks on different data types

int a = 4, b = 5;

std::cout << a + b << std::endl; // int + int // Addition

std::string h = "Hello", w = "World!";

std::cout << h + w << std::endl; // string + string // Concatenation
```

```
int a = 4, b = 5;

std::cout << a + b << std::endl; // int + int

std::string h = "Hello", w = "World!";

std::cout << h + w << std::endl; // string + string
```

Operator overloading

The compiler comes with a built-in version of the plus operator (+) for operands of basic data types.

Expression `a + b` becomes function call `operator+(a, b)` (`operator+` is the function name).

Similarly, `h + w` becomes function call `operator+(h, w)`.

Depending on the type of the arguments passed, the compiler will make a call to the appropriate overloaded function.

Resolving overloaded operators

When evaluating an expression containing an operator, the compiler uses the following rules:

- If all of the operands are fundamental data types,
 - The compiler will call a built-in routine if one exists. If one does not exist, the compiler will produce a compiler error.
- If any of the operands are user-defined data types (e.g. one of your classes, or an enum type),
 - The compiler looks to see whether the type has a matching overloaded operator function that it can call. If it can't find one, it will try to convert one or more of the user-defined type operands into fundamental data types so it can use a matching built-in operator. If that fails, then it will produce a compile error.

Why operator overloading?

- Operator overloading allows us to redefine the way operator works for user-defined types only.
- Operator overloading enables us to use notation closer to the target domain.
 - We can add two matrices by writing $M1 + M2$ rather than writing $M1.add(M2)$.
 - We can replace codes like `add(multiply(a, b), divide(a, b));` with $(a*b) + (a/b)$
- If we find ourselves limited by the way the C++ operators work, we can change term to do whatever we want.

Operator overloading

A form of polymorphism

All operators can be overloaded except

- Class member access operator (`.`, `.*`)
- Scope resolution operator (`::`)
- Size operator (`sizeof`)
- Conditional operator (`? :`)
- `typeid`, and
- the casting operators (e.g., `dynamic_cast`)

Operator overloading

Syntax:

```
returnType operator* (parameters) ;
```


Keyword operator symbol

Operators can be overloaded in 3 ways:

1. Using member functions
2. Using friend functions
3. Using normal functions

Overloading unary operators

Unary operators **act on only one operand.**

Examples:

increment operator (`++`), decrement operator (`--`), unary minus (`-`), not (`!`)

Overloading unary operators

Suppose we need to overload the increment operator (++) so that it increments seconds by 1.

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time() { }  
        Time(short hours, short minutes, short seconds)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
};
```

Overloading unary operators

- **When overloading an operator using a member function, one less argument than its number of operands is required.**
 - Why? Because one of the operand becomes the implicit `*this` object.
- Hence, unary operator requires no argument, i.e.
`returnType operator++ ();`
- If `c` is an object of a class with the overloaded `++` operator, the statement `c++;` will be equivalent to `c.operator++ ();`

Overloading unary operators

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time() { }  
        Time(short hours, short minutes, short seconds)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
  
        void operator++();  
};
```

Returns nothing

Overloaded function

Overloading unary operators

Implementing the member function that overloads the ++ operator

Operand is
the calling
object.

```
void Time::operator++() {  
    ++this->seconds;  
    // If seconds > 60, update minutes accordingly  
    this->minutes += this->seconds / 60;  
    this->seconds = this->seconds % 60;  
    // If minutes > 60, update hours accordingly  
    this->hours += this->minutes / 60;  
    this->minutes = this->minutes % 60;  
}
```

Overloading unary operators


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

int main() {
    Time t1(12, 30, 45);
    ++t1;

    Time t = ++t1;    // Will not work. Because the return type is void in the overloaded
function.
}
```

Overloading unary operators

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time() { }  
        Time(short hours, short minutes, short seconds)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
  
        Time operator++();  
};
```



Returns a Time object. This will enable statements like `Time t = ++t1;`

Overloading unary operators

```
Time Time::operator++() {  
    ++this->seconds;  
    // If seconds > 60, update minutes accordingly  
    this->minutes += this->seconds / 60;  
    this->seconds = this->seconds % 60;  
    // If minutes > 60, update hours accordingly  
    this->hours += this->minutes / 60;  
    this->minutes = this->minutes % 60;  
    return *this;  
}
```

```
#include <iostream>  
#include "Time.h"  
  
int main() {  
    Time t1(12, 30, 45);  
    ++t1;  
  
    Time t = ++t1;    // Works!!  
}
```

Overloading unary operators

Increment operator has two forms: postfix and prefix.

For both postfix and prefix increment operators, the overloaded function name would be the same (`operator++`). The number arguments would also be the same.

So how it is possible to differentiate the two when overloading?

Answer: C++ uses a “dummy variable” or “dummy argument” for the postfix operators.

Overloading unary operators

C++ uses a “dummy variable” or “dummy argument” for the postfix operators.

With `Time operator++();` we are basically overloading the prefix operator.

To overload the postfix operator, C++ uses a fake integer parameter that only serves to distinguish the postfix version of increment/decrement from the prefix version.

```
Time operator++(int);
```

Overloading unary operators

```
// For prefix increment operator
Time Time::operator++() {
    ++this->seconds;
    this->minutes += this->seconds / 60;
    this->seconds = this->seconds % 60;
    this->hours += this->minutes / 60;
    this->minutes = this->minutes % 60;
    // First increment the value and then
    // return the updated object
    return Time(hours, minutes, seconds);
}
```

Nameless temporary object



```
// For postfix increment operator
Time Time::operator++(int) {
    // Keep the values of the original object
    Time temp(hours, minutes, seconds);
    // Update the object
    this->seconds++;
    this->minutes += this->seconds / 60;
    this->seconds = this->seconds % 60;
    this->hours += this->minutes / 60;
    this->minutes = this->minutes % 60;
    // Return the original object
    return temp;
}
```

Lab 4 exercise

Question 1: Define a class, `Vector`, which represents either a column vector or a row vector. A column (row) vector is a matrix consisting of a single column (row) of m elements.

Let $m = 3$ in this program. (Use three member variables instead of an array.)

Overload the unary minus operator to negative all the elements in the vector.

For example, if $a = [2 \ -5 \ 6]$ is a row vector, then $-a$ must return $[-2 \ 5 \ -6]$.

Overloading binary operators

Binary operators can be overloaded just as easily as unary operators.

Binary operators act on two operands.

Examples:

Arithmetic operators (+, -, *, /, %), relational operators (==, >, < etc.),
assignment operator (=)

Overloading binary operators

When overloading an operator using a member function,

- The overloaded operator must be added as a member function of the left operand.
- The left operand becomes the implicit `*this` object
- All other operands become function parameters.

Overloading binary operators

When overloading an operator using a member function,


- The overloaded operator must be added as a member function of the left operand.
- The left operand becomes the implicit `*this` object
- All other operands become function parameters.

For example, in the statement `c = c1 + c2;` where `c`, `c1` and `c2` are objects of a class with overloaded `+` operator, `c1` will invoke the `operator+()` function with `c2` as an argument.

That is, `c = c1 + c2;` is equivalent to `c = c1.operator+(c2);`

Overloading binary operators

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time() { }  
        Time(short hours, short minutes, short seconds)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
  
        Time operator+(const Time &);  
};
```



Overloaded function

Overloading binary operators

```
Time Time::operator+(const Time &d1) {  
    Time d = d1;  
  
    d.seconds += this->seconds;  
    d.minutes += d.seconds / 60;  
    d.seconds %= 60;  
    d.minutes += this->minutes;  
    d.hours += d.minutes / 60;  
    d.minutes %= 60;  
    d.hours += this->hours;  
  
    return d;  
}
```

```
#include <iostream>  
#include "Time.h"  
  
int main() {  
    Time t1(12, 30, 45);  
    Time t2(10, 20, 25);  
  
    Time t = t1 + t2;  
}
```

Overloading binary operators using friend function

- A **friend function** is a function that can access the private members of a class as though it were a member of that class.
- A friend function may be either a normal function, or a member function of another class.
- To declare a friend function, simply use the `friend` keyword in front of the prototype of the function you wish to be a friend of the class.

friend returnType functionName(parameters);

Note that the `friend` keyword is placed only in the function declaration and not in the function definition.

Overloading binary operators using friend function

When overloading an operator using a friend function,

- The function requires one argument for a unary operator, and two arguments for a binary operator.
- The arguments may be passed either by value or reference.
- We can have a built-in data type as the left operand unlike when using a member function for operator overloading, where the left operand must always be an object of a class.

That is, the statement `c = 2 + c1;` will not work for a member function but for a friend function.

Overloading binary operators using friend function

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time() { }  
        Time(short hours, short minutes, short seconds)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
  
        friend Time operator+(const Time &, const Time &);  
};
```



A friend function

Overloading binary operators using friend function

```
Time operator+(const Time &d1, const Time &d2) {
```

```
    Time d = d1;
```

```
    d.seconds += d2.seconds;
```

```
    d.minutes += d.seconds / 60;
```

```
    d.seconds %= 60;
```

```
    d.minutes += d2.minutes;
```

```
    d.hours += d.minutes / 60;
```

```
    d.minutes %= 60;
```

```
    d.hours += d2.hours;
```

```
    return d;
```

```
}
```

Time:: is not needed because it is not a member function.
friend keyword is also not needed in the function definition.

```
#include <iostream>
```

```
#include "Time.h"
```

```
int main() {
```

```
    Time t1(12, 30, 45);
```

```
    Time t2(10, 20, 25);
```

```
    Time t = t1 + t2;
```

```
}
```

Overloading the I/O operators

In this program, how can we print the value of all member variables of the object `t`?

We could print each of the individual variables on the screen in the following way:

```
std::cout << t.getHours() << ":" << t.getMinutes() << ":" << t.getSeconds();
```

Here, `getHours()`, `getMinutes()` and `getSeconds()` are public member functions in class `Time`, which return the value of `hours`, `minutes`, and `seconds` respectively.

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

int main() {
    Time t1(12, 30, 45);
    Time t2(10, 20, 25);

    Time t = t1 + t2;
}
```

Tedious!!

Overloading the I/O operators

In this program, how can we print the value of all member variables of the object `t`?

We could have a `print()` method as a member function in class `Time`, and call this method when needed.

```
void Time::print() {  
    std::cout << hours << ":" << minutes << ":" << seconds;  
}
```

Still not a good solution because `print()` returns `void`, so it can't be called in the middle of an output statement.

```
std::cout << "Time = " << print(); // Doesn't work
```

```
#include <iostream>  
#include "Time.h"  
  
int main() {  
    Time t1(12, 30, 45);  
    Time t2(10, 20, 25);  
  
    Time t = t1 + t2;  
}
```


Overloading the I/O operators

In this program, how can we print the value of all member variables of the object `t`?

We can overload the `<<` operator and print the object `t` in the following way:

```
std::cout << "Time = " << t << std::endl;
```

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

int main() {
    Time t1(12, 30, 45);
    Time t2(10, 20, 25);

    Time t = t1 + t2;

}
```

Overloading the << operator

- Overloading `operator<<` is similar to overloading `operator+`, except that the parameter types are different.
- The `<<` operator is a binary operator.
- In the expression `std::cout << t;`,
 - The left operand is the `std::cout` object, and
 - The right operand is the object `t`.
- `std::cout` is actually an object of type `std::ostream`. Therefore, our overloaded function will look like this:

```
friend std::ostream& operator<<(std::ostream &outputStream, const Time &t);
```

Overloading the << operator

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time(short hours = 0, short minutes = 0, short seconds = 0)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
  
        friend std::ostream& operator<<(std::ostream &outputStream, const Time &t);  
};
```

Overloading the << operator

```
std::ostream& operator<<(std::ostream& outputStream, const Time& t) {  
    return outputStream << t.hours << ":" << t.minutes << ":" << t.seconds;  
}
```

```
#include <iostream>  
#include "Time.h"  
  
int main() {  
    Time t1(12, 30, 45);  
    std::cout << "Time = " << t1 << std::endl;  
}
```

The output will be:

Time = 12:30:45

Overloading the >> operator

The >> operator can be overloaded in the same way but the differences are

- The return type is `std::istream`
- The first argument is an object of type `std::istream`
- The second argument cannot be a `const`.

So, our function will look like this:

```
friend std::istream& operator>>(std::istream &inputStream, Time &t);
```

Overloading the >> operator

```
std::istream& operator>>(std::istream &inputStream, const Time &t) {  
    return inputStream >> t.hours >> t.minutes >> t.seconds;  
}
```

```
#include <iostream>  
#include "Time.h"  
  
int main() {  
    Time t1(12, 30, 45);  
    std::cout << "Please enter time in the order: hours minutes seconds" << std::endl;  
  
    std::cin >> t2;  
}
```

Limitations on operator overloading

- Not all operators can be overloaded, e.g. ::
- Only the existing operators can be overloaded. New operators cannot be created or existing operators cannot be renamed.
- At least one of the operands in an overloaded operator must be a user-defined type.
 - For example, overloading the plus operator to work with one integer and one double is not allowed.
- It is not possible to change the number of operands an operator supports.
 - That means a binary operator cannot be overloaded to work with three operands.
- All operators keep their default precedence and associativity (regardless of what they're used for) and this cannot be changed.
 - For example, if \wedge operator is overloaded to do the exponentiation, since operator \wedge has a lower precedence level than the basic arithmetic operators, expressions may be evaluated incorrectly.
 $2 + 3 \wedge 2$ will result in 25 instead of 11.

Overloading an operator using a normal function

Overloading an operator using a normal function is similar to using a friend function. The differences are

- The `friend` keyword is not needed.
- A normal function cannot access the private members of the operands. So, we may need getter methods in the calling object.

Overloading an operator using a normal function

```
class Time {  
    private:  
        short hours, minutes, seconds;  
    public:  
        Time(short hours = 0, short minutes = 0, short seconds = 0)  
            : hours(hours), minutes(minutes), seconds(seconds) { }  
  
        short getHours() const { return hours; }  
        short getMinutes() const { return minutes; }  
        short getSeconds() const { return seconds; }  
};
```

Overloading an operator using a normal function

```
// Neither a member function nor a friend function
Time operator-(const Time &d1, const Time &d2) {
    short seconds = d1.getSeconds();
    short minutes = d1.getMinutes();
    short hours = d1.getHours();

    if (seconds < d2.getSeconds()) { minutes--; seconds += 60; }
    seconds -= d2.getSeconds();

    if (minutes < d2.getMinutes()) { hours--; minutes += 60; }
    minutes -= d2.getMinutes();
    hours -= d2.getHours();

    return Time(hours, minutes, seconds);
}
```

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

Time operator-(const Time &d1, const Time &d2);

int main() {

    Time t1(12, 30, 45), t3(10, 35, 53);
    Time t4 = t3 - t1;

}
```

When to use a normal, friend, or member function overload

In most cases, the language leaves it up to you to determine whether you want to use the normal/friend or member function version of the overload.

The rules of thumb:

- If you're overloading assignment (`=`), subscript (`[]`), function call (`()`), or member selection (`->`), do so as a member function.
- If you're overloading a unary operator, do so as a member function.
- If you're overloading a binary operator that does not modify its left operand (e.g. `operator+`), do so as a normal function (preferred) or friend function.

When to use a normal, friend, or member function overload

The rules of thumb (contd.):

- If you're overloading a binary operator that modifies its left operand, but you can't modify the definition of the left operand (e.g. `operator<<`, which has a left operand of type `ostream`), do so as a normal function (preferred) or friend function.
- If you're overloading a binary operator that modifies its left operand (e.g. `operator+=`), and you can modify the definition of the left operand, do so as a member function.
- Prefer overloading operators as normal functions instead of friends if it's possible to do so without adding additional functions.

Lab 4 exercise

Question 2: In the class, `Vector`, defined in Question 1, overload the following operators:

1. `operator+` (Addition of a row-vector and a column-vector (or vice versa) is not allowed)
2. `operator-` (Subtraction of a row-vector from a column-vector (or vice versa) is not allowed)
3. `operator*` (A column-vector can multiply only a row-vector or vice versa!!)
4. `operator<<`
5. `operator>>`
6. `operator+=`
7. `operator==`
8. `operator>` (If `a` and `b` are two vectors of the same type (either row or column), then `a > b` if each element in `a` is greater than the corresponding element in `b`, i.e. `a.x > b.x` and `a.y > b.y` and `a.z > b.z`)

Copy assignment operator

Recall the **Rule of 3**:

If a class defines any of the following then it should probably explicitly define all three:

1. Destructor
2. Copy constructor
3. **Copy assignment operator**

Copy assignment operator

Copy assignment operator is basically overloading the assignment operator.

```
Time& operator=(const Time&);
```

A copy assignment is called when we assign an object to another object (of the same class).

```
Time t1, t2;  
t2 = t1;    // Assignment by copy assignment operator
```

Copy assignment operator

Assignment vs copy constructor

- If a new object has to be created before the copying can occur, the copy constructor is used.
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

```
Time t1, t2;  
t2 = t1; // Assignment by copy assignment operator  
  
Time t3(t1); // Initialization by copy constructor  
Time t4 = t1; // Initialization by copy constructor
```