```
13.5 — Overloading operators using member functions
♣ ALEX ● AUGUST 30, 2021
  In lesson 13.2 -- Overloading the arithmetic operators using friend functions, you learned how to overload the arithmetic operators using friend functions. You also
learned you can overload operators as normal functions. Many operators can be overloaded in a different way: as a member function.
Overloading operators using a member function is very similar to overloading operators using a friend function. 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.

As a reminder, here's how we overloaded operator+ using a friend function:
  1 | #include <iostream>
      class Cents
          int m_cents {};
      public:
          Cents(int cents)
              : m_cents { cents } { }
          // Overload Cents + int
 12
          friend Cents operator+(const Cents &cents, int value);
  14
          int getCents() const { return m_cents; }
 15
  16
      // note: this function is not a member function!
      Cents operator+(const Cents &cents, int value)
 20
          return Cents(cents.m_cents + value);
 21
  22
 24
      int main()
 25
          Cents cents1 { 6 };
          Cents cents2 { cents1 + 2 };
          std::cout << "I have " << cents2.getCents() << "</pre>
 29 cents.\n";
 31
       return 0;
Converting a friend overloaded operator to a member overloaded operator is easy:
1. The overloaded operator is defined as a member instead of a friend (Cents::operator+ instead of friend operator+)
2. The left parameter is removed, because that parameter now becomes the implicit *this object.
3. Inside the function body, all references to the left parameter can be removed (e.g. cents.m_cents becomes m_cents, which implicitly references the *this object).
Now, the same operator overloaded using the member function method:
  1 | #include <iostream>
      class Cents
   5 | private:
          int m_cents {};
      public:
          Cents(int cents)
              : m_cents { cents } { }
  10
          // Overload Cents + int
  12
 13
          Cents operator+ (int value);
 14
          int getCents() const { return m_cents; }
 15
  16
      // note: this function is a member function!
      // the cents parameter in the friend version is now the implicit *this parameter
      Cents Cents::operator+ (int value)
 21
  22
          return Cents { m_cents + value };
 23
 24
 25
      int main()
 26
          Cents cents1 { 6 };
 27
          Cents cents2 { cents1 + 2 };
          std::cout << "I have " << cents2.getCents() << " cents.\n";</pre>
  30
 31
          return 0;
 32
Note that the usage of the operator does not change (in both cases, cents1 + 2), we've simply defined the function differently. Our two-parameter friend function
becomes a one-parameter member function, with the leftmost parameter in the friend version (cents) becoming the implicit *this parameter in the member function version.
Let's take a closer look at how the expression cents1 + 2 evaluates.
In the friend function version, the expression cents1 + 2 becomes function call operator+(cents1, 2). Note that there are two function parameters. This is straightforward.
In the member function version, the expression cents1 + 2 becomes function call cents1.operator+(2). Note that there is now only one explicit function parameter, and
cents1 has become an object prefix. However, in lesson 12.10 -- The hidden "this" pointer, we mentioned that the compiler implicitly converts an object prefix into a hidden
leftmost parameter named *this. So in actuality, cents1.operator+(2) becomes operator+(¢s1, 2), which is almost identical to the friend version.
Both cases produce the same result, just in slightly different ways.
So if we can overload an operator as a friend or a member, which should we use? In order to answer that question, there's a few more things you'll need to know.
Not everything can be overloaded as a friend function
The assignment (=), subscript ([]), function call (()), and member selection (->) operators must be overloaded as member functions, because the language requires them to be.
Not everything can be overloaded as a member function
In lesson 13.4 -- Overloading the I/O operators, we overloaded operator << for our Point class using the friend function method. Here's a reminder of how we did that:
  1 | #include <iostream>
      class Point
  5 | private:
          double m_x {}, m_y {}, m_z {};
      public:
          Point(double x=0.0, double y=0.0, double z=0.0)
              : m_x { x }, m_y { y }, m_z { z }
  10
 11
  12
 13
          friend std::ostream& operator<< (std::ostream &out, const Point &point);</pre>
  14
 15
  16
      std::ostream& operator<< (std::ostream &out, const Point &point)</pre>
  18
          // Since operator<< is a friend of the Point class, we can access Point's members directly.
  20
          out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ")";
 21
 22
           return out;
 23
  24
      int main()
 26
 27
          Point point1 { 2.0, 3.0, 4.0 };
 29
          std::cout << point1;</pre>
  30
 31
          return 0;
 32
However, we are not able to overload operator << as a member function. Why not? Because the overloaded operator must be added as a member of the left operand. In this
case, the left operand is an object of type std::ostream. std::ostream is fixed as part of the standard library. We can't modify the class declaration to add the overload as a
member function of std::ostream.
This necessitates that operator<< be overloaded as a normal function (preferred) or a friend.
Similarly, although we can overload operator+(Cents, int) as a member function (as we did above), we can't overload operator+(int, Cents) as a member function, because int
isn't a class we can add members to.
Typically, we won't be able to use a member overload if the left operand is either not a class (e.g. int), or it is a class that we can't modify (e.g. std::ostream).
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. However, one of the
two is usually a better choice than the other.
When dealing with binary operators that don't modify the left operand (e.g. operator+), the normal or friend function version is typically preferred, because it works for all
parameter types (even when the left operand isn't a class object, or is a class that is not modifiable). The normal or friend function version has the added benefit of
"symmetry", as all operands become explicit parameters (instead of the left operand becoming *this and the right operand becoming an explicit parameter).
When dealing with binary operators that do modify the left operand (e.g. operator+=), the member function version is typically preferred. In these cases, the leftmost
operand will always be a class type, and having the object being modified become the one pointed to by *this is natural. Because the rightmost operand becomes an explicit
parameter, there's no confusion over who is getting modified and who is getting evaluated.
Unary operators are usually overloaded as member functions as well, since the member version has no parameters.
The following rules of thumb can help you determine which form is best for a given situation:
• 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.
 • If you're overloading a binary operator that modifies its left operand, but you can't add members to the class 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.
         Next lesson
          13.6 Overloading unary operators +, -, and !
         Back to table of contents
          Previous lesson
           13.4 Overloading the I/O operators
              B U URL INLINE CODE C++ CODE BLOCK HELP!
              Leave a comment...
           Name*
                                                                                                                                          POST COMMENT
                                                                                                          Notify me about replies: 💄
            @ Email*
           Avatars from https://gravatar.com/ are connected to your provided email
           address.
                                                                                                                                                    Newest ▼
   91 COMMENTS
           Jonas
          ( January 19, 2022 6:01 am
  Edit 1 Hour Later. It seems this phenomenon is called copy constructor.
 Could you explain what exactly is going on in line 3 of this code;
   1 | Cents Cents::operator+ (int value)
            return Cents { m_cents + value };
 I'm unsure if it creates an object of type 'Cents', which it then calls the constructor for (using our new integer), and then returns the object. But I dont think
 that makes sense, since our object 'cents2'
  1 | Cents cents2 { cents1 + 2 };
 would then be initialized with an object of type 'Cents', while it expects an integer(?).
 What am I missing? I appreciate the clarification.
 🕜 Last edited 1 month ago by Jonas
  0 Reply
                 Yes, the copy constructor is invoked here.
         ■ 1 Reply
           Steve
           ① December 30, 2021 7:40 am
 Why overloaded operators can not be a static member function of a class?
  1 0 → Reply
                 Reply to Steve ① December 30, 2021 7:18 pm
        What advantage would this have over using a free function?
         □ 0 Reply
                       Steve
                       Reply to Alex ① December 31, 2021 6:33 am
              I guess that you can declare it inside your class.h file and define it on your class.cpp. As a free function defining it inside your class.cpp file doesn't feel
              right. all free functions should go in a different cpp file. Correct me if I am wrong please, and thanks!
              1 0 → Reply
                             Alex Author
                             Reply to Steve ① December 31, 2021 10:40 am
                    It's fine (and expected) to have generic free functions that operate on a class type bundled into the class.h/cpp file.
                    Free functions that have application-specific logic should go in separate files.
                    1 0 → Reply
          © December 30, 2021 6:13 am
 Friend functions are able to access class data members directly. Isn't that faster than using normal functions which have to use getters and setters?
 🕜 Last edited 2 months ago by Rii
          Reply
  0
                 Reply to Rii ① December 30, 2021 7:12 pm
         Probably not, calls to getters and setters are likely to be converted by the compiler into direct member access.
         0 Reply
           Rekt0707
          © November 30, 2021 8:05 am
 1-When dealing with binary operators that don't modify the left operand (e.g. operator+), the normal or friend
 function version is typically preferred,
 When dealing with binary operators that do modify the left operand (e.g. operator+=), the member function version
 is typically preferred.
 2-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.
 If you're overloading a binary operator that modifies its left operand, but you can't add members to the class
 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.
  /*****
 So much confusion here.
 Could you explain it further or give links to examples?
  ******
 🗹 Last edited 3 months ago by Rekt0707
  ■ 0 Reply
          Taylor7500
          October 27, 2021 9:39 am
 Since you can't overload the IO operators as member functions, is there a best practice on where to define the (friend) operator<< and operator>> functions?
 Does it just go in the same file as the main() for each project, or can it be put somewhere else (e.g. the associated .cpp file for the object) and immediately
 reused elsewhere if I bring the object over to a different project?
  1 0 → Reply
                 Alex Author
                 Reply to Taylor7500 October 30, 2021 1:08 pm
        If the implementations are trivial, you can define them inside the class definition itself. If they are not trivial, you can define them in a .cpp file with the same
        base filename as the header. That way the .cpp and .h file travel as a pair, representing all of the code needed to use the class, and can be easily ported or
        shared between projects.
         0 Reply
           George
          © October 23, 2021 10:45 pm
 So why can't we make the << operator a member of the second operand?
          Reply
                 Alex Author
                 Reply to George ① October 25, 2021 3:26 pm
         Because with member operator overloads, the left operand becomes the implicit object.
        e.g. a + b would resolve to a.operator+(b).
        If b had an operator+, it doesn't get used in this case.
         0 Reply
           JamesC
          © August 30, 2021 3:05 am
 Not using member initialization list in the first two code examples.
  ■ 0 Reply
          Tejero Joshua
          © August 16, 2021 3:52 pm
 "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."
 how does operator<< modified and not modified the same time?
         Reply
  0
                  Alex Author
                 Reply to Tejero Joshua ( ) August 17, 2021 11:05 am
        Overloaded operator << takes a left hand operand of type std::ostream and a right hand operator of whatever type we want. std::ostream is modified when we
         print to it. However, because std::ostream is a standard library class, we can't modify the class definition. Using the word modified twice is confusing. I rewrote
        the sentence as: "If you're overloading a binary operator that modifies its left operand, but you can't add members to the class 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."
         ● 0 Reply
          Lesommeil
          ① July 27, 2021 12:37 am
  1 | std::ostream& operator<< (std::ostream& out);</pre>
 It can do something interesting like this.
  1 | point1 << std::cout;</pre>
 If I saw it as I don't know it, I will get panic.
         Reply
          Sarvottam Priyadarshee
  March 17, 2021 1:52 pm
 how string class is implemented?
 Specifically, My question is how do they know the length of the string before taking the input, I've encountered this problem while overloading operator>>
 and needed to take the input in MyString class!
 do they keep changing it, while taking the input, I mean the size of the string can be as large as 10^5 (maybe more)
 Even if they implement it using getchar() -> they must be going with a limit right, if they will allocate a very big size, then a lot of memory is wasted, space
```

std::cin has an input buffer that is used to hold characters as they are input. And yes, it does have a maximum size -- I'm not sure what that size is though. It may be implementation defined.

The point here is that the buffer is on the input stream, not the string, so only one is needed.

Proprietation

**Prop

complexity would increase!

Alex Author

1 0 → Reply