# Operator overloading

What are the basic rules and idioms for operator overloading in C++?

Note: The answers were given in *a specific order*, but since many users sort answers according to votes, rather than the time they were given, here's an ***index of the answers*** in the order in which they make most sense:

- The General Syntax of operator overloading in C++
- The Three Basic Rules of Operator Overloading in C++
- The Decision between Member and Non-member
- Common operators to overload
  - Assignment Operator
  - Input and Output Operators
  - Function call operator
  - Comparison operators
  - Arithmetic Operators
  - Array Subscripting
  - Operators for Pointer-like Types
  - Conversion Operators
- Overloading new and delete

*(Note: This is meant to be an entry to Stack Overflow's C++ FAQ. If you want to critique the idea of providing an FAQ in this form, then the posting on meta that started all this would be the place to do that. Answers to that question are monitored in the C++ chatroom, where the FAQ idea started out in the first place, so your answer is very likely to get read by those who came up with the idea.)*

c++ operators operator-overloading c++-faq

edited May 21 '14 at 5:47

Ben Voigt
**207k** 22 248 464

asked Dec 12 '10 at 12:44

sbi
**139k** 36 182 356

25 I should flag this for CW, but such a great work really needs to be awarded with some rep. – Matteo Italia Dec 12 '10 at 13:00

2 @Matteo: FWIW, writing this took the better part of my Sunday, an article draft that actually was meant to get published one day, and a big overdraw on my good-will credit with the kids (who wanted me to play LEGO with them instead). Please don't begrudge me the rep I get for it. – sbi Dec 12 '10 at 13:55

4 @sbi: maybe I expressed myself badly: I meant that in general somebody may flag this for CW, but you've done a great work, so I'm not going to push for CW at all, since you really deserve the rep you're getting from this. :) – Matteo Italia Dec 12 '10 at 14:43

44 If we're going to continue with the C++-FAQ tag, this is how entries should be formatted. – John Dibling Dec 14 '10 at 19:45

4 Oh my god! I haven't touched C++ for years and years (forced to write C# all day long now) and when I did, I was fairly young and clueless. I've decided to polish my C++ skills again, and posts like this... well, they're going to make it SO much easier! Thank you! – Tobias Jan 24 '13 at 22:14

7 Answers

## Common operators to overload

Most of the work in overloading operators is boiler-plate code. That is little wonder, since operators are merely syntactic sugar, their actual work could be done by (and often is forwarded to) plain functions. But it is important that you get this boiler-plate code right. If you fail, either your operator's code won't compile or your users' code won't compile or your users' code will behave surprisingly.

## Assignment Operator

There's a lot to be said about assignment. However, most of it has already been said in GMan's famous Copy-And-Swap FAQ, so I'll skip most of it here, only listing the perfect assignment operator for reference:

```
X& X::operator=(X rhs)
{
  swap(rhs);
  return *this;
}
```

## Bitshift Operators (used for Stream I/O)

The bitshift operators `<<` and `>>`, although still used in hardware interfacing for the bit-manipulation functions they inherit from C, have become more prevalent as overloaded stream input and output operators in most applications. For guidance overloading as bit-manipulation operators, see the section below on Binary Arithmetic Operators. For implementing your own custom format and parsing logic when your object is used with iostreams, continue.

The stream operators, among the most commonly overloaded operators, are binary infix operators for which the syntax specifies no restriction on whether they should be members or non-members. Since they change their left argument (they alter the stream's state), they should, according to the rules of thumb, be implemented as members of their left operand's type. However, their left operands are streams from the standard library, and while most of the stream output and input operators defined by the standard library are indeed defined as members of the stream classes, when you implement output and input operations for your own types, you cannot change the standard library's stream types. That's why you need to implement these operators for your own types as non-member functions. The canonical forms of the two are these:

```
std::ostream& operator<<(std::ostream& os, const T& obj)
{
  // write obj to stream

  return os;
}

std::istream& operator>>(std::istream& is, T& obj)
{
  // read obj from stream

  if( /* no valid object of T found in stream */ )
    is.setstate(std::ios::failbit);

  return is;
}
```

When implementing `operator>>`, manually setting the stream's state is only necessary when the reading itself succeeded, but the result is not what would be expected.

## Function call operator

The function call operator, used to create function objects, also known as functors, must be defined as a ***member*** function, so it always has the implicit `this` argument of member functions. Other than this it can be overloaded to take any number of additional arguments, including zero.

Throughout the C++ standard library, function objects are always copied. Your own function objects should therefore be cheap to copy. If a function object absolutely needs to use data which is expensive to copy, it is better to store that data elsewhere and have the function object refer to it.

## Comparison operators

The binary infix comparison operators should, according to the rules of thumb, be implemented as non-member functions[1]. The unary prefix negation `!` should (according to the same rules) be implemented as a member function. (but it is usually not a good idea to overload it.)

The standard library's algorithms (e.g. `std::sort()`) and types (e.g. `std::map`) will always only expect `operator<` to be present. However, the *users of your type will expect all the other operators to be present*, too, so if you define `operator<`, be sure to follow the third fundamental rule of operator overloading and also define all the other boolean comparison operators. The canonical way to implement them is this:

```
inline bool operator==(const X& lhs, const X& rhs){ /* do actual comparison */ }
inline bool operator!=(const X& lhs, const X& rhs){return !operator==(lhs,rhs);}
inline bool operator< (const X& lhs, const X& rhs){ /* do actual comparison */ }
inline bool operator> (const X& lhs, const X& rhs){return  operator< (rhs,lhs);}
inline bool operator<=(const X& lhs, const X& rhs){return !operator> (lhs,rhs);}
inline bool operator>=(const X& lhs, const X& rhs){return !operator< (lhs,rhs);}
```

The important thing to note here is that only two of these operators actually do anything, the others are just forwarding their arguments to either of these two to do the actual work.

The syntax for overloading the remaining binary boolean operators ( `||`, `&&` ) follows the rules of the comparison operators. However, it is *very* unlikely that you would find a reasonable use case for these[2].

[1] As with all rules of thumb, sometimes there might be reasons to break this one, too. If so, do not forget that the left-hand operand of the binary comparison operators, which for member functions will be `*this`, needs to be `const`, too. So a comparison operator implemented as a member function would have to have this signature:

```
bool operator<(const X& rhs) const { /* do actual comparison with *this */ }
```

(Note the `const` at the end.)

[2] It should be noted that the built-in version of `||` and `&&` use shortcut semantics. While the user defined ones (because they are syntactic sugar for method calls) do not use shortcut semantics. User will expect these operators to have shortcut semantics, and their code may depend on it, Therefore it is highly advised NEVER to define them.

## Arithmetic Operators

### Unary arithmetic operators

The unary increment and decrement operators come in both prefix and postfix flavor. To tell one from the other, the postfix variants take an additional dummy int argument. If you overload increment or decrement, be sure to always implement both prefix and postfix versions. Here is the canonical implementation of increment, decrement follows the same rules:

```
class X {
  X& operator++()
  {
    // do actual increment
    return *this;
  }
  X operator++(int)
  {
    X tmp(*this);
    operator++();
    return tmp;
  }
};
```

Note that the postfix variant is implemented in terms of prefix. Also note that postfix does an extra copy.[2]

Overloading unary minus and plus is not very common and probably best avoided. If needed, they should probably be overloaded as member functions.

[2] Also note that the postfix variant does more work and is therefore less efficient to use than the prefix variant. This is a good reason to generally prefer prefix increment over postfix increment. While compilers can usually optimize away the additional work of postfix increment for built-in types, they might not be able to do the same for user-defined types (which could be something as innocently looking as a list iterator). Once you got used to do `i++`, it becomes very hard to remember to do `++i` instead when `i` is not of a built-in type (plus you'd have to change code when changing a type), so it is better to make a habit of always using prefix increment, unless postfix is explicitly needed.

### Binary arithmetic operators

For the binary arithmetic operators, do not forget to obey the third basic rule operator overloading: If you provide `+`, also provide `+=`, if you provide `-`, do not omit `-=`, etc. Andrew Koenig is said to have been the first to observe that the compound assignment operators can be used as a base for their non-compound counterparts. That is, operator `+` is implemented in terms of `+=`, `-` is implemented in terms of `-=` etc.

According to our rules of thumb, `+` and its companions should be non-members, while their compound assignment counterparts (`+=` etc.), changing their left argument, should be a member. Here is the exemplary code for `+=` and `+`, the other binary arithmetic operators should be implemented in the same way:

```
class X {
  X& operator+=(const X& rhs)
  {
    // actual addition of rhs to *this
    return *this;
  }
};
inline X operator+(X lhs, const X& rhs)
{
  lhs += rhs;
  return lhs;
}
```

`operator+=` returns its result per reference, while `operator+` returns a copy of its result. Of course, returning a reference is usually more efficient than returning a copy, but in the case of `operator+`, there is no way around the copying. When you write `a + b`, you expect the result to be a new value, which is why `operator+` has to return a new value.[3] Also note that `operator+` takes its left operand ***by copy*** rather than by const reference. The reason for this is the same as the reason giving for `operator=` taking its argument per copy.

The bit manipulation operators `~ & | ^ << >>` should be implemented in the same way as the arithmetic operators. However, (except for overloading `<<` and `>>` for output and input) there are very few reasonable use cases for overloading these.

3 Again, the lesson to be taken from this is that `a += b` is, in general, more efficient than `a + b` and should be preferred if possible.

## Array Subscripting

The array subscript operator is a binary operator which must be implemented as a class member. It is used for container-like types that allow access to their data elements by a key. The canonical form of providing these is this:

```
class X {
        value_type& operator[](index_type idx);
  const value_type& operator[](index_type idx) const;
  // ...
};
```

Unless you do not want users of your class to be able to change data elements returned by `operator[]` (in which case you can omit the non-const variant), you should always provide both variants of the operator.

If value_type is known to refer to a built-in type, the const variant of the operator should return a copy instead of a const reference.

## Operators for Pointer-like Types

For defining your own iterators or smart pointers, you have to overload the unary prefix dereference operator `*` and the binary infix pointer member access operator `->`:

```
class my_ptr {
        value_type& operator*();
  const value_type& operator*() const;
        value_type* operator->();
  const value_type* operator->() const;
};
```

Note that these, too, will almost always need both a const and a non-const version. For the `->` operator, if `value_type` is of `class` (or `struct` or `union`) type, another `operator->()` is called recursively, until an `operator->()` returns a value of non-class type.

The unary address-of operator should never be overloaded.

For `operator->*()` see this question. It's rarely used and thus rarely ever overloaded. In fact, even iterators do not overload it.

Continue to Conversion Operators

edited Jun 14 '16 at 7:18

answered Dec 12 '10 at 12:47

sbi
**139k** 36 182 356

52 `operator->()` is actually *extremely* weird. It's not required to return a `value_type*` -- in fact, it can return another class type, **provided that class type has an `operator->()`**, which will then be called subsequently. This recursive calling of `operator->()`s proceeds until a `value_type*` return type occurs. Madness! :) – j_random_hacker Feb 22 '11 at 11:20

3 One comment: The implementation of binary arithmetic operators suggested is not such efficient as it can be. Se Boost operators headers simmetry note: boost.org/doc/libs/1_54_0/libs/utility/operators.htm#symmetry One more copy can be avoided if you use a local copy of the first parameter, do +=, and return the local copy. This enables NRVO optimization. – Manu343726 Jul 19 '13 at 8:36

1 As I mentioned in the chat, `L <= R` can also be expressed as `!(R < L)` instead of `!(L > R)`. Might save an extra layer of inlining in hard-to-optimize expressions (and it's also how Boost.Operators implements it). – TemplateRex Oct 10 '13 at 18:24

1 @thomthom: If a class doesn't have a publicly accessible API for getting at its state, you will have to either make everything that needs to access its state a member or a `friend` of the class. This, of course, is also true for all operators. – sbi Dec 4 '13 at 8:12

1 @mindandsky At least one of the arguments must be a user-defined type. Otherwise you're free, syntactically. But commonly it's overloaded for streams, that's why I used that as an example. – sbi May 20 '14 at 5:49

# The Three Basic Rules of Operator Overloading in C++

When it comes to operator overloading in C++, there are ***three basic rules you should follow***. As with all such rules, there are indeed exceptions. Sometimes people have deviated from them and the outcome was not bad code, but such positive deviations are few and far between. At the very least, 99 out of 100 such deviations I have seen were unjustified. However, it might just as well have been 999 out of 1000. So you'd better stick to the following rules.

1. ***Whenever the meaning of an operator is not obviously clear and undisputed, it should not be overloaded.*** *Instead, provide a function with a well-chosen name.* Basically, the first and foremost rule for overloading operators, at its very heart, says: *Don't do it*. That might seem strange, because there is a lot to be known about operator

overloading and so a lot of articles, book chapters, and other texts deal with all this. But despite this seemingly obvious evidence, *there are only a surprisingly few cases where operator overloading is appropriate*. The reason is that actually it is hard to understand the semantics behind the application of an operator unless the use of the operator in the application domain is well known and undisputed. Contrary to popular belief, this is hardly ever the case.

2. ***Always stick to the operator's well-known semantics.***
   C++ poses no limitations on the semantics of overloaded operators. Your compiler will happily accept code that implements the binary `+` operator to subtract from its right operand. However, the users of such an operator would never suspect the expression `a + b` to subtract `a` from `b`. Of course, this supposes that the semantics of the operator in the application domain is undisputed.
3. ***Always provide all out of a set of related operations.***
   *Operators are related to each other* and to other operations. If your type supports `a + b`, users will expect to be able to call `a += b`, too. If it supports prefix increment `++a`, they will expect `a++` to work as well. If they can check whether `a < b`, they will most certainly expect to also to be able to check whether `a > b`. If they can copy-construct your type, they expect assignment to work as well.

Continue to The Decision between Member and Non-member.

edited Jun 11 '13 at 7:13

Daniel Kamil Kozar
**9,107** 3 26 42

answered Dec 12 '10 at 12:45

sbi
**139k** 36 182 356

12 The only thing of which I am aware which violates any of these is `boost::spirit` lol. – Billy ONeal Dec 12 '10 at 17:50

54 @Billy: According to some, abusing `+` for string concatenation is a violation, but it has by now become well established praxis, so that it seems natural. Although I do remember a home-brew string class I saw in the 90ies that used binary `&` for this purpose (referring to BASIC for established praxis). But, yeah, putting it into the std lib basically set this in stone. The same goes for abusing `<<` and `>>` for IO, BTW. Why would left-shifting be the obvious output operation? Because we all learned about it when we saw our first "Hello, world!" application. And for no other reason. – sbi Dec 12 '10 at 19:56

4 @curiousguy: If you have to explain it, it's not obviously clear and undisputed. Likewise if you need to discuss or defend the overloading. – sbi Dec 2 '11 at 12:09

3 @sbi: "peer review" is always a good idea. To me a badly choosen operator is not different from a badly choosen function name (I saw many). Operator are just functions. No more no less. Rules are just the same. And to understand if an idea is good, the best way is understand how long does it takes to be understood. (Hence, peer review is a must, but peers must be chosen between people free from dogmas and prejudice.) – Emilio Garavaglia Apr 9 '12 at 16:57

3 @sbi To me, the only absolutely obvious and indisputable fact about `operator==` is that it should be an equivalence relation (IOW, you should not use non signaling NaN). There are many useful equivalence relations on containers. What does equality means? "`a` equals `b`" means that `a` and `b` have the same mathematical value. The concept of mathematical value of a (non-NaN) `float` is clear, but the mathematical value of a container can have many distinct (type recursive) useful definitions. The strongest definition of equality is "they are the same objects", and it is useless. – curiousguy Apr 10 '12 at 0:49

## The General Syntax of operator overloading in C++

You cannot change the meaning of operators for built-in types in C++, operators can only be overloaded for user-defined types[1]. That is, at least one of the operands has to be of a user-defined type. As with other overloaded functions, operators can be overloaded for a certain set of parameters only once.

Not all operators can be overloaded in C++. Among the operators that cannot be overloaded are: `.` `::` `sizeof` `typeid` `.*` and the only ternary operator in C++, `?:`

Among the operators that can be overloaded in C++ are these:

- arithmetic operators: `+` `-` `*` `/` `%` and `+=` `-=` `*=` `/=` `%=` (all binary infix); `+` `-` (unary prefix); `++` `--` (unary prefix and postfix)
- bit manipulation: `&` `|` `^` `<<` `>>` and `&=` `|=` `^=` `<<=` `>>=` (all binary infix); `~` (unary prefix)
- boolean algebra: `==` `!=` `<` `>` `<=` `>=` `||` `&&` (all binary infix); `!` (unary prefix)
- memory management: `new` `new[]` `delete` `delete[]`
- implicit conversion operators
- miscellany: `=` `[]` `->` `->*` `,` (all binary infix); `*` `&` (all unary prefix) `()` (function call, n-ary infix)

However, the fact that you *can* overload all of these does not mean you *should* do so. See the basic rules of operator overloading.

In C++, operators are overloaded in the form of ***functions with special names***. As with other functions, overloaded operators can generally be implemented either as a ***member function of their left operand's type*** or as ***non-member functions***. Whether you are free to choose or bound to use either one depends on several criteria.[2] A unary operator `@`[3], applied to an

object x, is invoked either as `operator@(x)` or as `x.operator@()`. A binary infix operator `@`, applied to the objects `x` and `y`, is called either as `operator@(x,y)` or as `x.operator@(y)`.[4]

Operators that are implemented as non-member functions are sometimes friend of their operand's type.

[1] The term "user-defined" might be slightly misleading. C++ makes the distinction between built-in types and user-defined types. To the former belong for example int, char, and double; to the latter belong all struct, class, union, and enum types, including those from the standard library, even though they are not, as such, defined by users.

[2] This is covered in a later part of this FAQ.

[3] The `@` is not a valid operator in C++ which is why I use it as a placeholder.

[4] The only ternary operator in C++ cannot be overloaded and the only n-ary operator must always be implemented as a member function.

Continue to The Three Basic Rules of Operator Overloading in C++.

edited Dec 12 '15 at 19:55

answered Dec 12 '10 at 12:46

sbi
**139k** 36 182 356

3 `%=` is not a "bit manipulation" operator – curiousguy Dec 1 '11 at 12:26

`~` is unary prefix, not binary infix. – mrkj Nov 2 '12 at 5:15

1 `.*` is missing from the list of non-overloadable operators. – celticminstrel Jul 4 '15 at 5:35

@celticminstrel: Indeed, and nobody noticed for 4.5 years... Thanks for pointing it out, I put it in. – sbi Jul 4 '15 at 19:56

*non-overloadable*! So, up there with `.` and `::`. And, it's probably because most people never use pointers-to-members. – celticminstrel Jul 4 '15 at 21:11

## The Decision between Member and Non-member

The binary operators `=` (assignment), `[]` (array subscription), `->` (member access), as well as the n-ary `()` (function call) operator, must always be implemented as ***member functions***, because the syntax of the language requires them to.

Other operators can be implemented either as members or as non-members. Some of them, however, usually have to be implemented as non-member functions, because their left operand cannot be modified by you. The most prominent of these are the input and output operators `<<` and `>>`, whose left operands are stream classes from the standard library which you cannot change.

For all operators where you have to choose to either implement them as a member function or a non-member function, ***use the following rules of thumb*** to decide:

1. If it is a ***unary operator***, implement it as a ***member*** function.
2. If a binary operator treats ***both operands equally*** (it leaves them unchanged), implement this operator as a ***non-member*** function.
3. If a binary operator does ***not*** treat both of its operands ***equally*** (usually it will change its left operand), it might be useful to make it a ***member*** function of its left operand's type, if it has to access the operand's private parts.

Of course, as with all rules of thumb, there are exceptions. If you have a type

```
enum Month {Jan, Feb, ..., Nov, Dec}
```

and you want to overload the increment and decrement operators for it, you cannot do this as a member functions, since in C++, enum types cannot have member functions. So you have to overload it as a free function. And `operator<()` for a class template nested within a class template is much easier to write and read when done as a member function inline in the class definition. But these are indeed rare exceptions.

(However, *if* you make an exception, do not forget the issue of `const`-ness for the operand that, for member functions, becomes the implicit `this` argument. If the operator as a non-member function would take its left-most argument as a `const` reference, the same operator as a member function needs to have a `const` at the end to make `*this` a `const` reference.)

Continue to Common operators to overload.

edited Feb 26 '15 at 8:40

answered Dec 12 '10 at 12:49

sbi
**139k** 36 182 356

Herb Sutter's item in Effective C++ (or is it C++ Coding Standards?) says one should prefer non-member non-friend functions to member functions, to increase the encapsulation of the class. IMHO, the encapsulation reason takes precedence to your rule of thumb, but it does not decrease the quality value of your rule of thumb. – paercebal Dec 12 '10 at 13:36

3 @paercebal: *Effective C++* is by Meyers, *C++ Coding Standards* by Sutter. Which one are you referring to? Anyway, I dislike the idea of, say, `operator+=()` not being a member. It has to change its left-hand operand, so by definition it has to dig deep into its innards. What would you gain by not making it a member? – sbi Dec 12 '10 at 13:39

6 @sbi: Item 44 in C++ Coding Standards (Sutter) **Prefer writing nonmember nonfriend functions**, of course, it only applies if you can actually write this function using only the public interface of the class. If you cannot (or can but it would hinder performance badly), then you have to make it either member or friend. – Matthieu M. Dec 12 '10 at 15:45

2 @sbi : Oops, Effective, Exceptional... No wonder I mix the names up. Anyway the gain is to limit as much as possible the number of functions that have access to an object private/protected data. This way, you increase the encapsulation of your class, making its maintenance/testing/evolution easier. – paercebal Dec 12 '10 at 16:51

9 @sbi : One example. Let's say you're coding a String class, with both the `operator +=` and the `append` methods. The `append` method is more complete, because you can append a substring of the parameter from index i to index n -1: `append(string, start, end)` It seems logical to have `+=` call append with `start = 0` and `end = string.size` . At that moment, append could be a member method, but `operator +=` doesn't need to be a member, and making it a non-member would decrease the quantity of code playing with the String innards, so it is a good thing.... ^_^ ... – paercebal Dec 12 '10 at 16:58

---

## Conversion Operators (also known as User Defined Conversions)

In C++ you can create conversion operators, operators that allow the compiler to convert between your types and other defined types. There are two types of conversion operators, implicit and explicit ones.

### Implicit Conversion Operators (C++98/C++03 and C++11)

An implicit conversion operator allows the compiler to implicitly convert (like the conversion between `int` and `long` ) the value of a user-defined type to some other type.

The following is a simple class with an implicit conversion operator:

```
class my_string {
public:
  operator const char*() const {return data_;} // This is the conversion operator
private:
  const char* data_;
};
```

Implicit conversion operators, like one-argument constructors, are user-defined conversions. Compilers will grant one user-defined conversion when trying to match a call to an overloaded function.

```
void f(const char*);

my_string str;
f(str); // same as f( str.operator const char*() )
```

At first this seems very helpful, but the problem with this is that the implicit conversion even kicks in when it isn't expected to. In the following code, `void f(const char*)` will be called because `my_string()` is not an lvalue, so the first does not match:

```
void f(my_string&);
void f(const char*);

f(my_string());
```

Beginners easily get this wrong and even experienced C++ programmers are sometimes surprised because the compiler picks an overload they didn't suspect. These problems can be mitigated by explicit conversion operators.

### Explicit Conversion Operators (C++11)

Unlike implicit conversion operators, explicit conversion operators will never kick in when you don't expect them to. The following is a simple class with an explicit conversion operator:

```
class my_string {
public:
  explicit operator const char*() const {return data_;}
private:
  const char* data_;
};
```

Notice the `explicit` . Now when you try to execute the unexpected code from the implicit conversion operators, you get a compiler error:

```
prog.cpp: In function 'int main()':
prog.cpp:15:18: error: no matching function for call to 'f(my_string)'
prog.cpp:15:18: note: candidates are:
prog.cpp:11:10: note: void f(my_string&)
```

```
prog.cpp:11:10: note:   no known conversion for argument 1 from ‘my_string’ to
‘my_string&’
prog.cpp:12:10: note: void f(const char*)
prog.cpp:12:10: note:   no known conversion for argument 1 from ‘my_string’ to
‘const char*’
```

To invoke the explicit cast operator, you have to use `static_cast`, a C-style cast, or a constructor style cast ( i.e. `T(value)` ).

However, there is one exception to this: The compiler is allowed to implicitly convert to `bool`. In addition, the compiler is not allowed to do another implicit conversion after it converts to `bool` (a compiler is allowed to do 2 implicit conversions at a time, but only 1 user-defined conversion at max).

Because the compiler will not cast "past" `bool`, explicit conversion operators now remove the need for the Safe Bool idiom. For example, smart pointers before C++11 used the Safe Bool idiom to prevent conversions to integral types. In C++11, the smart pointers use an explicit operator instead because the compiler is not allowed to implicitly convert to an integral type after it explicitly converted a type to bool.

Continue to Overloading `new` and `delete`.

edited Jan 7 '16 at 14:20

YSC
**4,705** 2 11 37

answered May 17 '13 at 18:32

JKor
**2,816** 3 15 32

1 Thank you very much, I consider this a fantastic addition to this FAQ! I have incorporated it into the TOC. I also made a very few small changes to your answer, hopefully improving it a bit by doing so. Feel free to roll them back if you disagree. – sbi May 21 '13 at 9:49

2 Thank you so much for the opportunity for me to add this to the FAQ! – JKor May 21 '13 at 22:10

---

## Overloading `new` and `delete`

***Note:*** This only deals with the ***syntax*** of overloading `new` and `delete`, not with the ***implementation*** of such overloaded operators. I think that the semantics of overloading ***`new` and `delete` deserve their own FAQ***, within the topic of operator overloading I can never do it justice.

### Basics

In C++, when you write a ***new expression*** like `new T(arg)` two things happen when this expression is evaluated: First ***`operator new`*** is invoked to obtain raw memory, and then the appropriate constructor of `T` is invoked to turn this raw memory into a valid object. Likewise, when you delete an object, first its destructor is called, and then the memory is returned to `operator delete`.
C++ allows you to tune both of these operations: memory management and the construction/destruction of the object at the allocated memory. The latter is done by writing constructors and destructors for a class. Fine-tuning memory management is done by writing your own `operator new` and `operator delete`.

The first of the basic rules of operator overloading – *don't do it* – applies especially to overloading `new` and `delete`. Almost the only reasons to overload these operators are ***performance problems*** and ***memory constraints***, and in many cases, other actions, like *changes to the algorithms* used, will provide a much ***higher cost/gain ratio*** than attempting to tweak memory management.

The C++ standard library comes with a set of predefined `new` and `delete` operators. The most important ones are these:

```
void* operator new(std::size_t) throw(std::bad_alloc);
void  operator delete(void*) throw();
void* operator new[](std::size_t) throw(std::bad_alloc);
void  operator delete[](void*) throw();
```

The first two allocate/deallocate memory for an object, the latter two for an array of objects. If you provide your own versions of these, they will ***not overload, but replace*** the ones from the standard library.
If you overload `operator new`, you should always also overload the matching `operator delete`, even if you never intend to call it. The reason is that, if a constructor throws during the evaluation of a new expression, the run-time system will return the memory to the `operator delete` matching the `operator new` that was called to allocate the memory to create the object in. If you do not provide a matching `operator delete`, the default one is called, which is almost always wrong.
If you overload `new` and `delete`, you should consider overloading the array variants, too.

### Placement `new`

C++ allows new and delete operators to take additional arguments.
So-called placement new allows you to create an object at a certain address which is passed to:

```
class X { /* ... */ };
char buffer[ sizeof(X) ];
```

```
void f()
{
  X* p = new(buffer) X(/*...*/);
  // ...
  p->~X(); // call destructor
}
```

The standard library comes with the appropriate overloads of the new and delete operators for this:

```
void* operator new(std::size_t,void* p) throw(std::bad_alloc);
void  operator delete(void* p,void*) throw();
void* operator new[](std::size_t,void* p) throw(std::bad_alloc);
void  operator delete[](void* p,void*) throw();
```

Note that, in the example code for placement new given above, `operator delete` is never called, unless the constructor of X throws an exception.

You can also overload `new` and `delete` with other arguments. As with the additional argument for placement new, these arguments are also listed within parentheses after the keyword `new`. Merely for historical reasons, such variants are often also called placement new, even if their arguments are not for placing an object at a specific address.

## Class-specific new and delete

Most commonly you will want to fine-tune memory management because measurement has shown that instances of a specific class, or of a group of related classes, are created and destroyed often and that the default memory management of the run-time system, tuned for general performance, deals inefficiently in this specific case. To improve this, you can overload new and delete for a specific class:

```
class my_class {
  public:
    // ...
    void* operator new();
    void  operator delete(void*,std::size_t);
    void* operator new[](size_t);
    void  operator delete[](void*,std::size_t);
    // ...
};
```

Overloaded thus, new and delete behave like static member functions. For objects of `my_class`, the `std::size_t` argument will always be `sizeof(my_class)`. However, these operators are also called for dynamically allocated objects of ***derived classes***, in which case it might be greater than that.

## Global new and delete

To overload the global new and delete, simply replace the pre-defined operators of the standard library with our own. However, this is rarely ever needs to be done.

edited Mar 23 '12 at 8:42

answered Dec 12 '10 at 13:07

sbi
**139k** 36 182 356

3 -1 for the title which contradicts itself. You're confusing people. The standard forms don't get overloaded, they get replaced as you say. – Yttrill Dec 12 '10 at 15:07

6 I also don't agree that replacing the global operator new and delete is usually for performance: on the contrary, it's usually for bug tracing. – Yttrill Dec 12 '10 at 15:14

You should also note, that if you use an overloaded new operator you're required to also provide a delete operator with matching arguments. You say that in the section on global new/delete where it isn't of much interest. – Yttrill Dec 12 '10 at 15:17

8 @Yttrill you are confusing things. The *meaning* gets overloaded. What "operator overloading" means is that the meaning is overloaded. It does not mean that literally functions are overloaded, and *in particular* operator new will not overload the Standard's version. @sbi doesn't claim the opposite. It's common to call it "overloading new" much as it is common to say "overloading addition operator". – Johannes Schaub - litb Dec 13 '10 at 4:51

19 @Yttrill sadly I accidentally upvoted your comment. What i really wanted to do is to downvote it :) Please take no offense :) – Johannes Schaub - litb Dec 13 '10 at 4:52

---

## Why can't `operator<<` function for streaming objects to `std::cout` or to a file be a member function?

Let's say you have:

```
struct Foo
{
   int a;
   double b;

   std::ostream& operator<<(std::ostream& out) const
   {
      return out << a << " " << b;
   }
};
```

Given that, you cannot use:

```
Foo f = {10, 20.0};
std::cout << f;
```

Since `operator<<` is overloaded as a member function of `Foo`, the LHS of the operator must be a `Foo` object. Which means, you will be required to use:

```
Foo f = {10, 20.0};
f << std::cout
```

which is very non-intuitive.

If you define it as a non-member function,

```
struct Foo
{
   int a;
   double b;
};

std::ostream& operator<<(std::ostream& out, Foo const& f)
{
   return out << f.a << " " << f.b;
}
```

You will be able to use:

```
Foo f = {10, 20.0};
std::cout << f;
```

which is very intuitive.

answered Jan 22 '16 at 19:00

R Sahu
**113k** 9 53 114

---

**protected** by Community ♦ Dec 10 '14 at 15:21

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?