|  |  |
| --- | --- |
|  | **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](http://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom), 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 functions1. 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 these2.  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](http://stackoverflow.com/q/8777845/140719). It's rarely used and thus rarely ever overloaded. In fact, even iterators do not overload it.  **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](http://stackoverflow.com/questions/3601602/what-are-rvalues-lvalues-xvalues-glvalues-and-prvalues), 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](http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Safe_bool). 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.  **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](http://stackoverflow.com/questions/7149461/), 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. |