Overloading operators

Comp Sci 1570 Introduction to C++

Missouri S&T | Computer Science

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Overloading operators

- Function overloading provides a mechanism to create and resolve function calls to multiple functions with the same name, so long as each function has a unique function prototype.
- This allows you to create variations of a function to work with different data types, without having to think up a unique name for each variant.
- In C++, operators are implemented as functions.
- By using function overloading on the operator functions, you can define your own versions of the operators that work with different data types (including classes that you've written).
- Using function overloading to overload operators is called operator overloading.

```
ReturnType classname(optional) :: operator OperatorSymbol (argument list)
{
  \\ statements;
}
```

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Rules for overloading operators

- You cannot create new operators.
- If you overload an operator, at least one of the parameters must be a user-defined type. Thus, you cannot redefine an operator for a built-in type.
- There are some operators you are prevented from overloading (see tables upcoming)
- You cannot change the arity (number of operands required) of a operator.
- You cannot change the order of precedence or associativity of operators by overloading them.
- An overloaded operator cannot have default arguments.

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Operators can be overloaded in several locations

- As a member function
- As a normal non-member function
- As a friend non-member function

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Operator+

- See code implementing it all 3 ways

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Overloading as a member function

- For example, the assignment $(=)$, subscript $([])$, function call $(())$, and member selection $(->)$ operators must be overloaded as member functions

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Overload as non-member and Non-member friend

- For example, we are not able to overload *operator* $<<$ () as a member function.
- 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 don't modify the class declaration to add the overload as a member function of std::ostream.

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Definitions
Rules
Basics
Methods of operator overload
Basic examples
Member
Non-member and non-member friend
When to use each
Overview of operators
Arithmetic
Comparison and relational
Logical
Bitwise
Compound assignment
Member and pointer
Other

When to use normal, friend, or member overload

- 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 to use normal, friend, or member overload

- 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.

When to use normal, friend, or member overload

- Unary operators are usually overloaded as member functions as well, since the member version has no parameters.

Summary for which method

- 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 modifies its left operand (e.g. operator+=), do so as a member function if you can.
- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function or friend function.
- **Remember:** Which method and syntax you use for each operator depends on the particular arbitrary implementation in C++, and requires checking/memorizing the rules

Outline

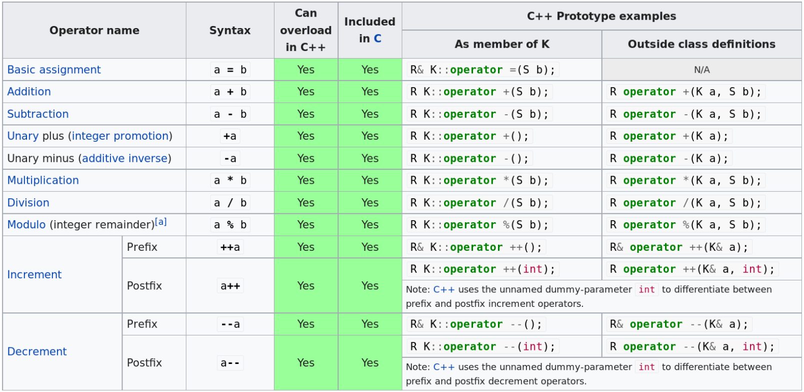
1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Arithmetic operators

- Here, *a*, *b*, and *c* represent valid values (literals, values from variables, or return value), object names, or lvalues, as appropriate.
- *R*, *S*, and *T*, stand for any type(*s*), and *K* for a class type or enumerated type.

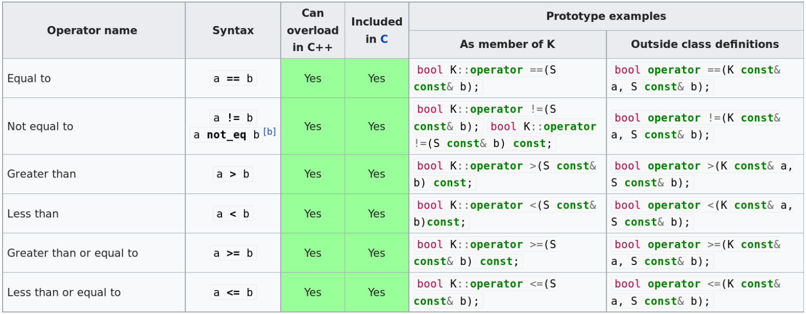


Operator name		Syntax	Can overload in C++	Included in C	C++ Prototype examples	
					As member of K	Outside class definitions
Basic assignment		`a = b`	Yes	Yes	`R& K::operator =(S b);`	N/A
Addition		`a + b`	Yes	Yes	`R K::operator +(S b);`	`R operator +(K a, S b);`
Subtraction		`a - b`	Yes	Yes	`R K::operator -(S b);`	`R operator -(K a, S b);`
Unary plus (integer promotion)		`+a`	Yes	Yes	`R K::operator +();`	`R operator +(K a);`
Unary minus (additive inverse)		`-a`	Yes	Yes	`R K::operator -();`	`R operator -(K a);`
Multiplication		`a * b`	Yes	Yes	`R K::operator *(S b);`	`R operator *(K a, S b);`
Division		`a / b`	Yes	Yes	`R K::operator /(S b);`	`R operator /(K a, S b);`
Modulo (integer remainder)[a]		`a % b`	Yes	Yes	`R K::operator %(S b);`	`R operator %(K a, S b);`
Increment	Prefix	`++a`	Yes	Yes	`R& K::operator ++();`	`R& operator ++(K& a);`
	Postfix	`a++`	Yes	Yes	`R K::operator ++(int);`	`R operator ++(K& a, int);`
					Note: C++ uses the unnamed dummy-parameter `int` to differentiate between prefix and postfix increment operators.	
Decrement	Prefix	`--a`	Yes	Yes	`R& K::operator --();`	`R& operator --(K& a);`
	Postfix	`a--`	Yes	Yes	`R K::operator --(int);`	`R operator --(K& a, int);`
					Note: C++ uses the unnamed dummy-parameter `int` to differentiate between prefix and postfix decrement operators.	

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Comparison and relational operators



Operator name	Syntax	Can overload in C++	Included in C	Prototype examples	
				As member of K	Outside class definitions
Equal to	`a == b`	Yes	Yes	`bool K::operator ==(S const& b);`	`bool operator ==(K const& a, S const& b);`

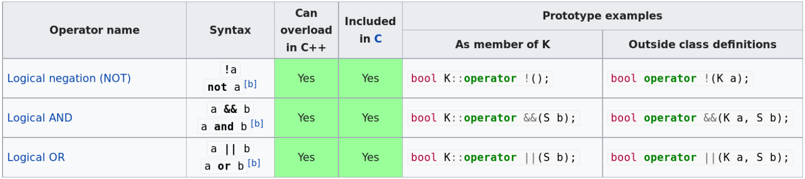
Not equal to	`a != b`
	`a not_eq b` [b]
		Yes	Yes	`bool K::operator !=(S const& b);`
				`bool K::operator !=(S const& b) const;`
					`bool operator !=(K const& a, S const& b);`

Greater than	`a > b`	Yes	Yes	`bool K::operator >(S const& b) const;`	`bool operator >(K const& a, S const& b);`
Less than	`a < b`	Yes	Yes	`bool K::operator <(S const& b)const;`	`bool operator <(K const& a, S const& b);`
Greater than or equal to	`a >= b`	Yes	Yes	`bool K::operator >=(S const& b) const;`	`bool operator >=(K const& a, S const& b);`
Less than or equal to	`a <= b`	Yes	Yes	`bool K::operator <=(S const& b);`	`bool operator <=(K const& a, S const& b);`

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Logical operators

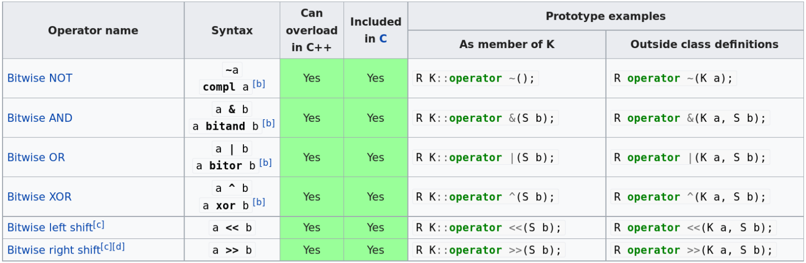


Operator name	Syntax	Can overload in C++	Included in C	Prototype examples	
				As member of K	Outside class definitions
Logical negation (NOT)	`!a` `not a` [b]	Yes	Yes	`bool K::operator !();`	`bool operator !(K a);`
Logical AND	`a && b` `a and b` [b]	Yes	Yes	`bool K::operator &&(S b);`	`bool operator &&(K a, S b);`
Logical OR	`a || b` `a or b` [b]	Yes	Yes	`bool K::operator ||(S b);`	`bool operator ||(K a, S b);`

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Bitwise operators



Operator name	Syntax	Can overload in C++	Included in C	Prototype examples	
				As member of K	Outside class definitions
Bitwise NOT	`~a` `compl a` [b]	Yes	Yes	`R K::operator ~();`	`R operator ~(K a);`
Bitwise AND	`a & b` `a bitand b` [b]	Yes	Yes	`R K::operator &(S b);`	`R operator &(K a, S b);`
Bitwise OR	`a | b` `a bitor b` [b]	Yes	Yes	`R K::operator |(S b);`	`R operator |(K a, S b);`
Bitwise XOR	`a ^ b` `a xor b` [b]	Yes	Yes	`R K::operator ^(S b);`	`R operator ^(K a, S b);`
Bitwise left shift[c]	`a << b`	Yes	Yes	`R K::operator <<(S b);`	`R operator <<(K a, S b);`
Bitwise right shift[c][d]	`a >> b`	Yes	Yes	`R K::operator >>(S b);`	`R operator >>(K a, S b);`

Outline

1 Definitions

2 Rules

- Basics

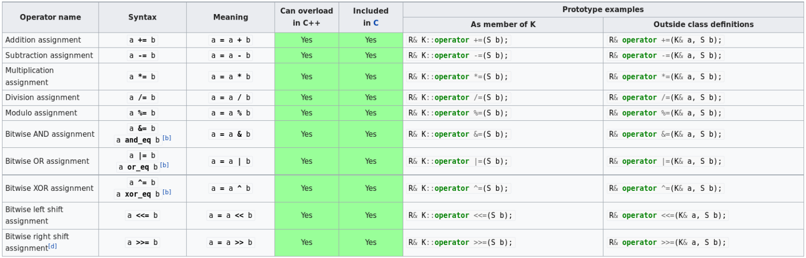
3 Methods of operator overload

- Basic examples
- Member
- Non-member and non-member friend
- When to use each

4 Overview of operators

- Arithmetic
- Comparison and relational
- Logical
- Bitwise
- Compound assignment
- Member and pointer
- Other

Compound assignment operators

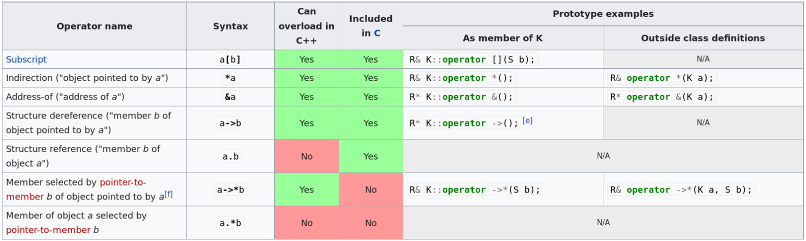


Operator name	Syntax	Meaning	Can overload in C++	Included in C	Prototype examples	
					As member of K	Outside class definitions
Addition assignment	`a += b`	`a = a + b`	Yes	Yes	`R& K::operator +=(S b);`	`R& operator +=(K& a, S b);`
Subtraction assignment	`a -= b`	`a = a - b`	Yes	Yes	`R& K::operator -=(S b);`	`R& operator -=(K& a, S b);`
Multiplication assignment	`a *= b`	`a = a * b`	Yes	Yes	`R& K::operator *=(S b);`	`R& operator *=(K& a, S b);`
Division assignment	`a /= b`	`a = a / b`	Yes	Yes	`R& K::operator /=(S b);`	`R& operator /=(K& a, S b);`
Modulo assignment	`a %= b`	`a = a % b`	Yes	Yes	`R& K::operator %=(S b);`	`R& operator %=(K& a, S b);`
Bitwise AND assignment	`a &= b` `a and_eq b` [b]	`a = a & b`	Yes	Yes	`R& K::operator &=(S b);`	`R& operator &=(K& a, S b);`
Bitwise OR assignment	`a |= b` `a or_eq b` [b]	`a = a | b`	Yes	Yes	`R& K::operator |=(S b);`	`R& operator |=(K& a, S b);`
Bitwise XOR assignment	`a ^= b` `a xor_eq b` [b]	`a = a ^ b`	Yes	Yes	`R& K::operator ^=(S b);`	`R& operator ^=(K& a, S b);`
Bitwise left shift assignment	`a <<= b`	`a = a << b`	Yes	Yes	`R& K::operator <<=(S b);`	`R& operator <<=(K& a, S b);`
Bitwise right shift assignment[d]	`a >>= b`	`a = a >> b`	Yes	Yes	`R& K::operator >>=(S b);`	`R& operator >>=(K& a, S b);`

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Member and pointer operators

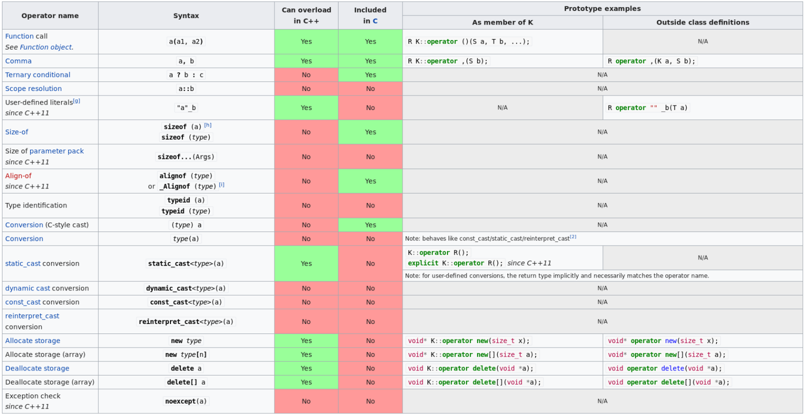


Operator name	Syntax	Can overload in C++	Included in C	Prototype examples	
				As member of K	Outside class definitions
Subscript	`a[b]`	Yes	Yes	`R& K::operator [](S b);`	N/A
Indirection ("object pointed to by *a*")	`*a`	Yes	Yes	`R& K::operator *();`	`R& operator *(K a);`
Address-of ("address of *a*")	`&a`	Yes	Yes	`R* K::operator &();`	`R* operator &(K a);`
Structure dereference ("member *b* of object pointed to by *a*")	`a->b`	Yes	Yes	`R* K::operator ->();` [e]	N/A
Structure reference ("member *b* of object *a*")	`a.b`	No	Yes	N/A	
Member selected by pointer-to-member *b* of object pointed to by *a*[f]	`a->*b`	Yes	No	`R& K::operator ->*(S b);`	`R& operator ->*(K a, S b);`
Member of object *a* selected by pointer-to-member *b*	`a.*b`	No	No	N/A	

Outline

1. Definitions
2. Rules
   - Basics
3. Methods of operator overload
   - Basic examples
   - Member
   - Non-member and non-member friend
   - When to use each
4. Overview of operators
   - Arithmetic
   - Comparison and relational
   - Logical
   - Bitwise
   - Compound assignment
   - Member and pointer
   - Other

Other operators



Operator name	Syntax	Can overload in C++	Included in C	Prototype examples	
				As member of K	Outside class definitions
Function call *See Function object.*	a(a1, a2)	Yes	Yes	R K::operator ()(S a, T b, ...);	N/A
Comma	a, b	Yes	Yes	R K::operator ,(S b);	R operator ,(K a, S b);
Ternary conditional	a ? b : c	No	Yes	N/A	
Scope resolution	a::b	No	No	N/A	
User-defined literals[g] *since C++11*	"a"_b	Yes	No	N/A	R operator "" _b(T a)
Size-of	**sizeof** (a) [h] **sizeof** (*type*)	No	Yes	N/A	
Size of parameter pack *since C++11*	**sizeof...**(Args)	No	No	N/A	
Align-of *since C++11*	**alignof** (*type*) or **_Alignof** (*type*) [i]	No	Yes	N/A	
Type identification	**typeid** (a) **typeid** (*type*)	No	No	N/A	
Conversion (C-style cast)	(*type*) a	No	Yes	N/A	
Conversion	*type*(a)	No	No	Note: behaves like const_cast/static_cast/reinterpret_cast[2]	
static_cast conversion	**static_cast**<*type*>(a)	Yes	No	K::operator R(); explicit K::operator R(); *since C++11*	N/A
				Note: for user-defined conversions, the return type implicitly and necessarily matches the operator name.	
dynamic cast conversion	**dynamic_cast**<*type*>(a)	No	No	N/A	
const_cast conversion	**const_cast**<*type*>(a)	No	No	N/A	
reinterpret_cast conversion	**reinterpret_cast**<*type*>(a)	No	No	N/A	
Allocate storage	**new** *type*	Yes	No	void* K::operator new(size_t x);	void* operator new(size_t x);
Allocate storage (array)	**new** *type*[n]	Yes	No	void* K::operator new[](size_t a);	void* operator new[](size_t a);
Deallocate storage	**delete** a	Yes	No	void K::operator delete(void *a);	void operator delete(void *a);
Deallocate storage (array)	**delete[]** a	Yes	No	void K::operator delete[](void *a);	void operator delete[](void *a);
Exception check *since C++11*	**noexcept**(a)	No	No	N/A	