Other operators

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
function call	`a(a1, a2)`	Yes	`R T::operator()(Arg1 &a1, Arg2 &a2, ... ...);`	N/A
comma	`a, b`	Yes	`T2& T::operator,(T2 &b);`	`T2& operator,(const T &a, T2 &b);`
conditional operator	`a ? b : c`	No	N/A	N/A

Explanation

The *function call* operator provides function semantics for any object.

The *conditional operator* (colloquially referred to as *ternary conditional*) checks the boolean value of the first expression and, depending on the resulting value, evaluates and returns either the second or the third expression.

Built-in function call operator

The function call expressions have the form

E **(** *A1*, *A2*, *A3*,... **)**

where

- *E* is an expression that names a function
- *A1*, *A2*, *A3*,... is a possibly empty list of arbitrary expressions or braced-init-lists (since C++11), except the comma operator is not allowed at the top level to avoid ambiguity.

The expression that names the function can be

a) lvalue expression that refers to a function
b) pointer to function
c) explicit class member access expression that selects a member function
d) implicit class member access expression, e.g. member function name used within another member function.

The function (or member) name specified by E can be overloaded, overload resolution rules used to decide which overload is to be called.

If E specifies a member function, it may be virtual, in which case the final overrider of that function will be called, using dynamic dispatch at runtime.

To call the function,

The expression E as well as all expressions A1, A2, A3, etc, provided as arguments are evaluated in arbitrary order, unsequenced with respect to each other.	(until C++17)
The expression E is sequenced before each of the expressions A1, A2, A3 as well as default arguments, if any. The argument expressions are evaluated in arbitrary order, indeterminately sequenced with respect to each other.	(since C++17)

Each function parameter is initialized with its corresponding argument after implicit conversion if necessary. If there is no corresponding argument, the corresponding default argument is used, and if there is none, the program is ill-formed. If the call is made to a member function, then the `this` pointer to current object is converted as if by explicit cast to the `this` pointer expected by the function. The initialization and destruction of each parameter occurs in the context of the caller, which means, for example, that if constructor of a parameter throws an exception, the exception handlers defined within the function, even as a function-try block, are not considered. If the function is a variadic function, default argument promotions are applied to all arguments matched by the ellipsis parameter. It is implementation-defined whether the lifetime of a parameter ends when the function in which it is defined returns or at the end of the enclosing full-expression.

The return type of a function call expression is the return type of the chosen function, decided using static binding (ignoring the `virtual` keyword), even if the overriding function that's actually called returns a different type. This allows the overriding functions to return pointers or references to classes that are derived from the return type returned by the base function, i.e. C++ supports covariant return types . If E specifies a destructor, the return type is `void`.

When an object of class type X is passed to or returned from a function, if each copy constructor, move constructor, and destructor of X is either trivial or deleted, and X has at least one non-deleted copy or move constructor, implementations are permitted to create a temporary object to hold the function parameter or result object. The temporary object is constructed from the function argument or return value, respectively, and the function's parameter or return object is initialized as if by using the non-deleted trivial constructor to copy the temporary (even if that constructor is inaccessible or would not be selected by overload resolution to perform a copy or move of the object). This allows objects of small class types, such as `std::complex` or `std::span`, to be passed to or returned from functions in registers.	(since C++17)

The value category of a function call expression is lvalue if the function returns an lvalue reference or an rvalue reference to function, is an xvalue if the function returns an rvalue reference to object, and is a prvalue otherwise. If the function call expression is a prvalue of object type, it must have complete type except when the prvalue is not materialized, such as (since C++17) when used as the operand of `decltype` (or as the right operand of a built-in comma operator expression that is the operand of `decltype`).

Function call expression is similar in syntax to value initialization `T()`, to function-style cast expression `T(A1)`, and to direct initialization of a temporary `T(A1, A2, A3, ...)`, where T is the name of a type.

Run this code

```
#include <cstdio>

struct S
{
    int f1(double d)
    {
        return printf("%f \n", d); // variable argument function call
    }

    int f2()
    {
        return f1(7); // member function call, same as this->f1()
                      // integer argument converted to double
    }
};

void f()
{
    puts("function called"); // function call
}

int main()
{
    f();    // function call
    S s;
    s.f2(); // member function call
}
```

Output:

```
function called
7.000000
```

Built-in comma operator

The comma operator expressions have the form

E1 , *E2*

In a comma expression `E1, E2`, the expression E1 is evaluated, its result is discarded (although if it has class type, it won't be destroyed until the end of the containing full expression), and its side effects are completed before evaluation of the expression E2 begins (note that a user-defined `operator,` cannot guarantee sequencing) (until C++17).

The type, value, and value category of the result of the comma expression are exactly the type, value, and value category of the second operand, E2. If E2 is a temporary expression (since C++17), the result of the expression is that temporary expression (since C++17). If E2 is a bit-field, the result is a bit-field.

The comma in various comma-separated lists, such as function argument lists (`f(a, b, c)`) and initializer lists `int a[] = {1, 2, 3}`, is not the comma operator. If the comma operator needs to be used in such contexts, it has to be parenthesized: `f(a, (n++, n + b), c)`

Using an unparenthesized comma expression as second (right) argument of a subscript operator is deprecated. (since C++20) (until C++23)

For example, `a[b, c]` is deprecated and `a[(b, c)]` is not.

An unparenthesized comma expression cannot be second (right) argument of a subscript operator. For example, `a[b, c]` is either ill-formed or equivalent to `a.operator[](b, c)`. (since C++23)

Parentheses are needed to for using a comma expression as the subscript, e.g., `a[(b, c)]`.

Run this code

```
#include <iostream>

int main()
{
    int n = 1;
    int m = (++n, std::cout << "n = " << n << '\n', ++n, 2 * n);
    std::cout << "m = " << (++m, m) << '\n';
}
```

Output:

```
n = 2
m = 7
```

Conditional operator

The conditional operator expressions have the form

E1 **?** *E2* **:** *E3*

The first operand of the conditional operator is evaluated and contextually converted to `bool`. After both the value evaluation and all side effects of the first operand are completed, if the result was `true`, the second operand is evaluated. If the result was `false`, the third operand is evaluated.

The type and value category of the conditional expression `E1 ? E2 : E3` are determined according to the following rules:

1) If either E2 or E3 has type `void`, then one of the following must be true, or the program is ill-formed:

1.1) Either E2 or E3 (but not both) is a (possibly parenthesized) throw-expression. The result of the conditional operator has the type and the value category of the other expression. If the other expression is a bit-field, the result is a bit-field. Such conditional operator was commonly used in C++11 constexpr programming prior to C++14.

```
std::string str = 2 + 2 == 4 ? "OK" : throw std::logic_error("2 + 2 != 4");
```

1.2) Both E2 and E3 are of type `void` (including the case when they are both throw-expressions). The result is a prvalue of type `void`.

```
2 + 2 == 4 ? throw 123 : throw 456;
```

2) Otherwise, if E2 or E3 are glvalue bit-fields of the same value category and of types cv1 T and cv2 T, respectively, the operands are considered to be of type cv T for the remainder of this section, where cv is the union of cv1 and cv2. (since C++14)

3) Otherwise, if E2 and E3 have different types, at least one of which is a (possibly cv-qualified) class type, or both are glvalues of the same value category and have the same type except for cv-qualification, then an attempt is made to form an implicit conversion sequence ignoring member access, whether an operand is a bit-field, or whether a conversion function is deleted (since C++14) from each of the operands to the *target type* determined by the other operand, as described below. An operand (call it X) of type TX can be converted to the *target type* of the other operand (call it Y) of type TY as follows:

3.1) If Y is an lvalue, the target type is TY&, and the reference must bind directly to an lvalue;

3.2) If Y is an xvalue, the target type is TY&&, and the reference must bind directly;

3.3) If Y is a prvalue, or if neither the above conversion sequences can be formed and at least one of TX and TY is a (possibly cv-qualified) class type,

3.3.1) if TX and TY are the same class type (ignoring cv-qualification) and TY is at least as cv-qualified as TX, the target type is TY,

3.3.2) otherwise, if TY is a base class of TX, the target type is TY with the cv-qualifiers of TX:

```
struct A {};

struct B : A {};

using T = const B;

A a = true ? A() : T(); // Y = A(), TY = A, X = T(), TX = const B. Target = const A
```

3.3.3) otherwise, the target type is the type that Y would have after applying the lvalue-to-rvalue, array-to-pointer, and function-to-pointer standard conversions

3.4) If both sequences can be formed (E2 to target type of E3 and E3 to target type of E2), or only one can be formed but it is the ambiguous conversion sequence, the program is ill-formed.

3.5) If exactly one conversion sequence can be formed (note that it may still be ill-formed e.g. due to access violation), that conversion sequence is applied and the converted operand is used in place of the original operand for the remained of this description (starting at (4))

3.6) If no conversion sequence can be formed, the operands are left unchanged for the remainder of this description

4) If E2 and E3 are glvalues of the same type and the same value category, then the result has the same type and value category, and is a bit-field if at least one of E2 and E3 is a bit-field.

5) Otherwise, the result is a prvalue. If E2 and E3 do not have the same type, and either has (possibly cv-qualified) class type, overload resolution is performed using the built-in candidates below to attempt to convert the operands to built-in types. If the overload resolution fails, the program is ill-formed. Otherwise, the selected conversions are applied and the converted operands are used in place of the original operands for step 6.

6) The lvalue-to-rvalue, array-to-pointer, and function-to-pointer conversions are applied to the second and third operands. Then,

6.1) If both E2 and E3 now have the same type, the result is a prvalue of that type designating a temporary object (until C++17) whose result object is (since C++17) copy-initialized from whatever operand was selected after evaluating E1.

6.2) If both E2 and E3 have arithmetic or enumeration type: the *usual arithmetic conversions* are applied to bring them to *common type*, and that type is the result.

6.3) If both E2 and E3 are pointers, or one is a pointer and the other is a null pointer constant, then pointer conversions and qualification conversions are applied to bring them to common type, and that type is the result.

```
int* intPtr;

using Mixed = decltype(true ? nullptr : intPtr);

static_assert(std::is_same_v<Mixed, int*>); // nullptr becoming int*
```

6.4) If both E2 and E3 are pointers to members, or one is a pointer to member and the other is a null pointer constant, then pointer-to-member conversions and qualification conversions are applied to bring them to common type, and that type is the result.

```
struct A
{
    int* m_ptr;
} a;

int* A::* memPtr = &A::m_ptr; // memPtr is a pointer to member m_ptr of A

// memPtr makes nullptr as type of pointer to member m_ptr of A
static_assert(std::is_same_v<decltype(false ? memPtr : nullptr), int*A::*>);

// a.*memPtr is now just pointer to int and nullptr also becomes pointer to int
static_assert(std::is_same_v<decltype(false ? a.*memPtr : nullptr), int*>);
```

6.5) If both E2 and E3 are null pointer constants, and at least one of which is of type std::nullptr_t, then the result's type is std::nullptr_t.

6.6) In all other cases, the program is ill-formed.

This section is incomplete
Reason: any chance to make this more readable without losing the fine point? At the very least, a one-line micro-example for each bullet point would help tremendously

For every pair of promoted arithmetic types L and R and for every type P, where P is a pointer, pointer-to-member, or scoped enumeration type, the following function signatures participate in the overload resolution performed in step 5 of the rules above:

```
LR operator?:(bool, L, R);
P operator?:(bool, P, P);
```

where LR is the result of usual arithmetic conversions performed on L and R. The operator “?:” cannot be overloaded, these function signatures only exist for the purpose of overload resolution.

The return type of a conditional operator is also accessible as the binary type trait std::common_type.

Run this code

```
#include <string>
#include <iostream>

struct Node
{
    Node* next;
    int data;

    // deep-copying copy constructor
    Node(const Node& other)
      : next(other.next ? new Node(*other.next) : NULL)
      , data(other.data)
    {}

    Node(int d) : next(NULL), data(d) {}

    ~Node() { delete next ; }
};

int main()
{
    // simple rvalue example
    int n = 1 > 2 ? 10 : 11;  // 1 > 2 is false, so n = 11

    // simple lvalue example
    int m = 10;
    (n == m ? n : m) = 7; // n == m is false, so m = 7

    //output the result
    std::cout << "n = " << n << "\nm = " << m;
}
```

Output:

```
n = 11
m = 7
```

Standard library

Many classes in the standard library overload `operator()` to be used as function objects.

operator()	deletes the object or array (public member function of std::default_delete<T>)
operator()	returns the sum of two arguments (public member function of std::plus<T>)
operator()	returns the difference between two arguments (public member function of std::minus<T>)
operator()	returns the product of two arguments (public member function of std::multiplies<T>)
operator()	returns the result of the division of the first argument by the second argument (public member function of std::divides<T>)
operator()	returns the remainder from the division of the first argument by the second argument (public member function of std::modulus<T>)
operator()	returns the negation of the argument (public member function of std::negate<T>)
operator()	checks if the arguments are equal (public member function of std::equal_to<T>)
operator()	checks if the arguments are not equal (public member function of std::not_equal_to<T>)
operator()	checks if the first argument is greater than the second (public member function of std::greater<T>)
operator()	checks if the first argument is less than the second (public member function of std::less<T>)
operator()	checks if the first argument is greater than or equal to the second (public member function of std::greater_equal<T>)
operator()	checks if the first argument is less than or equal to the second (public member function of std::less_equal<T>)
operator()	returns the logical AND of the two arguments (public member function of std::logical_and<T>)
operator()	returns the logical OR of the two arguments (public member function of std::logical_or<T>)
operator()	returns the logical NOT of the argument (public member function of std::logical_not<T>)
operator()	returns the result of bitwise AND of two arguments (public member function of std::bit_and<T>)
operator()	returns the result of bitwise OR of two arguments (public member function of std::bit_or<T>)
operator()	returns the result of bitwise XOR of two arguments (public member function of std::bit_xor<T>)
operator()	returns the logical complement of the result of a call to the stored predicate (public member function of std::unary_negate<Predicate>)
operator()	returns the logical complement of the result of a call to the stored predicate (public member function of std::binary_negate<Predicate>)
operator()	calls the stored function (public member function of std::reference_wrapper<T>)
operator()	invokes the target (public member function of std::function<R(Args...)>)
operator()	lexicographically compares two strings using this locale's collate facet (public member function of std::locale)
operator()	compares two values of type `value_type` (public member function of std::map<Key,T,Compare,Allocator>::value_compare)
operator()	compares two values of type `value_type` (public member function of std::multimap<Key,T,Compare,Allocator>::value_compare)
operator()	executes the function (public member function of std::packaged_task<R(Args...)>)
operator() (C++11)	advances the engine's state and returns the generated value (public member function of std::linear_congruential_engine<UIntType,a,c,m>)
operator() (C++11)	generates the next random number in the distribution (public member function of std::uniform_int_distribution<IntType>)

The comma operator is not overloaded by any class in the standard library. The boost library uses `operator,` in boost.assign (http://www.boost.org/doc/libs/release/libs/assign/doc/index.html#intro) , boost.spirit, and other libraries. The database access library SOCI (http://soci.sourceforge.net/doc.html) also overloads `operator,` .

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 446 (https://cplusplus.github.io/CWG/issues/446.html)	C++98	it was unspecified whether a temporary is created for an lvalue-to-rvalue conversion on the conditional operator	always creates a temporary if the operator returns a class rvalue
CWG 462 (https://cplusplus.github.io/CWG/issues/462.html)	C++98	if the second operand of a comma expression is a temporary, it was unspecified whether its lifetime will be extended when the result of the comma expression is bound to a reference	the result of the comma expression is the temporary in this case (hence its lifetime is extended)
CWG 1550 (https://cplusplus.github.io/CWG/issues/1550.html)	C++98	parenthesized throw-expression not allowed in ?: if other operand is non-void	parenthesized throw-expressions accepted

CWG 1560 (https://cplusplus.github.io/CWG/issues/1560.html)	C++98	void operand in ?: caused gratuitous l-to-r conversion on the other operand, always resulting in rvalue	?: with a void can be lvalue
CWG 1932 (https://cplusplus.github.io/CWG/issues/1932.html)	C++14	same-type bit-fields were missing in ?:	handled by underlying types
CWG 1895 (https://cplusplus.github.io/CWG/issues/1895.html)	C++14	unclear if deleted or inaccessible conversion function prevents conversion in ?:, and conversions from base class to derived class prvalue were not considered	handled like overload resolution

See also

Operator precedence Operator overloading

Common operators						
assignment	increment decrement	arithmetic	logical	comparison	member access	**other**
a = b a += b a -= b a *= b a /= b a %= b a &= b a |= b a ^= b a <<= b a >>= b	++a --a a++ a--	+a -a a + b a - b a * b a / b a % b ~a a & b a | b a ^ b a << b a >> b	!a a && b a || b	a == b a != b a < b a > b a <= b a >= b a <=> b	a[b] *a &a a->b a.b a->*b a.*b	a(...) a, b a ? b : c

Special operators

static_cast converts one type to another related type
dynamic_cast converts within inheritance hierarchies
const_cast adds or removes cv qualifiers
reinterpret_cast converts type to unrelated type
C-style cast converts one type to another by a mix of static_cast, const_cast, and reinterpret_cast
new creates objects with dynamic storage duration
delete destructs objects previously created by the new expression and releases obtained memory area
sizeof queries the size of a type
sizeof... queries the size of a parameter pack (since C++11)
typeid queries the type information of a type
noexcept checks if an expression can throw an exception (since C++11)
alignof queries alignment requirements of a type (since C++11)

C documentation for **Other operators**