Implicit conversions

Implicit conversions are performed whenever an expression of some type T1 is used in context that does not accept that type, but accepts some other type T2; in particular:

- when the expression is used as the argument when calling a function that is declared with T2 as parameter;
- when the expression is used as an operand with an operator that expects T2;
- when initializing a new object of type T2, including `return` statement in a function returning T2;
- when the expression is used in a `switch` statement (T2 is integral type);
- when the expression is used in an `if` statement or a loop (T2 is `bool`).

The program is well-formed (compiles) only if there exists one unambiguous *implicit conversion sequence* from T1 to T2.

If there are multiple overloads of the function or operator being called, after the implicit conversion sequence is built from T1 to each available T2, overload resolution rules decide which overload is compiled.

Note: in arithmetic expressions, the destination type for the implicit conversions on the operands to binary operators is determined by a separate set of rules, *usual arithmetic conversions*

Order of the conversions

Implicit conversion sequence consists of the following, in this order:

1) zero or one *standard conversion sequence*;
2) zero or one *user-defined conversion*;
3) zero or one *standard conversion sequence* (only if a user-defined conversion is used).

When considering the argument to a constructor or to a user-defined conversion function, only a standard conversion sequence is allowed (otherwise user-defined conversions could be effectively chained). When converting from one non-class type to another non-class type, only a standard conversion sequence is allowed.

A standard conversion sequence consists of the following, in this order:

1) zero or one conversion from the following set: *lvalue-to-rvalue conversion*, *array-to-pointer conversion*, and *function-to-pointer conversion*;
2) zero or one *numeric promotion* or *numeric conversion*;
3) zero or one *function pointer conversion*; (since C++17)
4) zero or one *qualification conversion*.

A user-defined conversion consists of zero or one non-explicit single-argument converting constructor or non-explicit conversion function call

An expression e is said to be *implicitly convertible to T2* if and only if T2 can be copy-initialized from e, that is the declaration `T2 t = e;` is well-formed (can be compiled), for some invented temporary `t`. Note that this is different from direct initialization (`T2 t(e)`), where explicit constructors and conversion functions would additionally be considered.

Contextual conversions

In the following contexts, the type `bool` is expected and the implicit conversion is performed if the declaration `bool t(e);` is well-formed (that is, an explicit conversion function such as `explicit T::operator bool() const;` is considered). Such expression e is said to be *contextually converted to bool*.

- the controlling expression of `if`, `while`, `for`;
- the operands of the built-in logical operators `!`, `&&` and `||`;
- the first operand of the conditional operator `?:`; (since C++11)
- the predicate in a `static_assert` declaration;
- the expression in a `noexcept` specifier;
- the expression in an `explicit` specifier; (since C++20)

In the following contexts, a context-specific type T is expected, and the expression e of class type E is only allowed if E has a single non-explicit user-defined conversion function to an allowable type (until C++14)there is exactly one type T among the allowable types such that E has non-explicit conversion functions whose return types are (possibly cv-qualified) T or reference to (possibly cv-qualified) T, and e is implicitly convertible to T (since C++14). Such expression e is said to be *contextually implicitly converted* to the specified type T. Note that explicit conversion functions are not considered, even though they are considered in contextual conversions to bool. (since C++11)

- the argument of the delete-expression (T is any object pointer type);
- integral constant expression, where a literal class is used (T is any integral or unscoped (since C++11) enumeration type, the selected user-defined conversion function must be constexpr);
- the controlling expression of the `switch` statement (T is any integral or enumeration type).

```
#include <cassert>

template<typename T>
class zero_init
{
    T val;
public:
    zero_init() : val(static_cast<T>(0)) {}
    zero_init(T val) : val(val) {}
    operator T&() { return val; }
    operator T() const { return val; }
};

int main()
{
    zero_init<int> i;
    assert(i == 0);

    i = 7;
    assert(i == 7);

    switch (i) {}     // error until C++14 (more than one conversion function)
                      // OK since C++14 (both functions convert to the same type int)
    switch (i + 0) {} // always okay (implicit conversion)
}
```

Value transformations

Value transformations are conversions that change the value category of an expression. They take place whenever an expression appears as an operand of an operator that expects an expression of a different value category.

Lvalue to rvalue conversion

A glvalue of any non-function, non-array type T can be implicitly converted to a prvalue of the same type. If T is a non-class type, this conversion also removes cv-qualifiers.

The object denoted by the glvalue is not accessed if:

- the conversion occurs in an unevaluated context, such as an operand of `sizeof`, `noexcept`, `decltype`, (since C++11) or the static form of `typeid`
- the glvalue has the type `std::nullptr_t`: in this case the resulting prvalue is the null pointer constant `nullptr`. (since C++11)
- the value stored in the object is a compile-time constant and certain other conditions are satisfied (see ODR-use)

If T is a non-class type, the value contained in the object is produced as the prvalue result. For a class type, this conversion

effectively copy-constructs a temporary object of type T using the original glvalue as the constructor argument, and that temporary object is returned as a prvalue. (until C++17)

converts the glvalue to a prvalue whose result object is copy-initialized by the glvalue. (since C++17)

This conversion models the act of reading a value from a memory location into a CPU register.

If the object to which the glvalue refers contains an indeterminate value (such as obtained by default initializing a non-class automatic variable), the behavior is undefined except if the indeterminate value is of possibly cv-qualified `unsigned char` or `std::byte` (since C++17) type.

The behavior is also implementation-defined (rather than undefined) if the glvalue contains a pointer value that was invalidated.

Array to pointer conversion

An lvalue or rvalue of type "array of N T" or "array of unknown bound of T" can be implicitly converted to a prvalue of type "pointer to T". If the array is a prvalue, temporary materialization occurs. (since C++17) The resulting pointer refers to the first element of the array (see array to pointer decay for details)

Temporary materialization

A prvalue of any complete type T can be converted to an xvalue of the same type T. This conversion initializes a temporary object of type T from the prvalue by evaluating the prvalue with the temporary object as its result object, and produces an xvalue denoting the temporary object. If T is a class or array of class type, it must have an accessible and non-deleted destructor.

```
struct S { int m; };
int i = S().m; // member access expects glvalue as of C++17;
               // S() prvalue is converted to xvalue
```

Temporary materialization occurs in the following situations: (since C++17)

- when binding a reference to a prvalue;
- when performing a member access on a class prvalue;
- when performing an array-to-pointer conversion (see above) or subscripting on an array prvalue;
- when initializing an object of type `std::initializer_list<T>` from a braced-init-list;
- when `typeid` is applied to a prvalue (this is part of an unevaluated expression);
- when `sizeof` is applied to a prvalue (this is part of an unevaluated expression);
- when a prvalue appears as a discarded-value expression.

Note that temporary materialization does *not* occur when initializing an object from a prvalue of the same type (by direct-initialization or copy-initialization): such object is initialized directly from the initializer. This ensures "guaranteed copy elision".

Function to pointer

An lvalue of function type T can be implicitly converted to a prvalue pointer to that function. This does not apply to non-static member functions because lvalues that refer to non-static member functions do not exist.

Numeric promotions

Integral promotion

prvalues of small integral types (such as `char`) may be converted to prvalues of larger integral types (such as `int`). In particular, arithmetic operators do not accept types smaller than `int` as arguments, and integral promotions are automatically applied after lvalue-to-rvalue conversion, if applicable. This conversion always preserves the value.

The following implicit conversions are classified as integral promotions:

- **`signed char`** or **`signed short`** can be converted to `int`;
- **`unsigned char`** or **`unsigned short`** can be converted to `int` if it can hold its entire value range, and `unsigned int` otherwise;
- **`char`** can be converted to `int` or `unsigned int` depending on the underlying type: `signed char` or `unsigned char` (see above);
- **`wchar_t`**, **`char8_t`** (since C++20), **`char16_t`**, and **`char32_t`** (since C++11) can be converted to the first type from the following list able to hold their entire value range: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long` (since C++11);
- an unscoped (since C++11) enumeration type whose underlying type is not fixed can be converted to the first type from the following list able to hold their entire value range: `int`, `unsigned int`, `long`, `unsigned long`, `long long`, or `unsigned long long`, extended integer types with higher conversion rank (in rank order, signed given preference over unsigned) (since C++11), if the value range is greater, no integral promotions apply;
- an unscoped (since C++11) enumeration type whose underlying type is fixed can be converted to its underlying type, and, if the underlying type is also subject to integral promotion, to the promoted underlying type. Conversion to the unpromoted underlying type is better for the purposes of overload resolution;
- a bit-field type can be converted to `int` if it can represent entire value range of the bit-field, otherwise to `unsigned int` if it can represent entire value range of the bit-field, otherwise no integral promotions apply;
- the type `bool` can be converted to `int` with the value `false` becoming `0` and `true` becoming `1`.

Note that all other conversions are not promotions; for example, overload resolution chooses `char` -> `int` (promotion) over `char` -> `short` (conversion).

Floating-point promotion

A prvalue of type `float` can be converted to a prvalue of type `double`. The value does not change.

Numeric conversions

Unlike the promotions, numeric conversions may change the values, with potential loss of precision.

Integral conversions

A prvalue of an integer type or of an unscoped (since C++11) enumeration type can be converted to any other integer type. If the conversion is listed under integral promotions, it is a promotion and not a conversion.

- If the destination type is unsigned, the resulting value is the smallest unsigned value equal to the source value modulo 2^n where n is the number of bits used to represent the destination type.

  That is, depending on whether the destination type is wider or narrower, signed integers are sign-extended[1] or truncated and unsigned integers are zero-extended or truncated respectively.
- If the destination type is signed, the value does not change if the source integer can be represented in the destination type. Otherwise the result is implementation-defined (until C++20)the unique value of the destination type equal to the source value modulo 2^n where n is the number of bits used to represent the destination type. (since C++20). (Note that this is different from signed integer arithmetic overflow, which is undefined).
- If the source type is `bool`, the value `false` is converted to zero and the value `true` is converted to the value one of the destination type (note that if the destination type is `int`, this is an integer promotion, not an integer conversion).
- If the destination type is `bool`, this is a boolean conversion (see below).

Floating-point conversions

A prvalue of a floating-point type can be converted to a prvalue of any other floating-point type. If the conversion is listed under floating-point promotions, it is a promotion and not a conversion.

- If the source value can be represented exactly in the destination type, it does not change.
- If the source value is between two representable values of the destination type, the result is one of those two values (it is implementation-defined which one, although if IEEE arithmetic is supported, rounding defaults to nearest).
- Otherwise, the behavior is undefined.

Floating-integral conversions

- A prvalue of floating-point type can be converted to a prvalue of any integer type. The fractional part is truncated, that is, the fractional part is discarded. If the value cannot fit into the destination type, the behavior is undefined (even when the destination type is unsigned, *modulo* arithmetic does not apply). If the destination type is `bool`, this is a boolean conversion (see below).
- A prvalue of integer or unscoped (since C++11) enumeration type can be converted to a prvalue of any floating-point type. The result is exact if possible. If the value can't fit into the destination type but cannot be represented exactly, it is implementation defined whether the closest higher or the closest lower representable value will be selected, although if IEEE arithmetic is supported, rounding defaults to nearest. If the value cannot fit into the destination type, the behavior is undefined. If the source type is `bool`, the value `false` is converted to zero, and the value `true` is converted to one.

Pointer conversions

- A *null pointer constant* (see `NULL`), can be converted to any pointer type, and the result is the null pointer value of that type. Such conversion (known as *null pointer conversion*) is allowed to convert to a cv-qualified type as a single conversion, that is, it's not considered a combination of numeric and qualifying conversions.
- A prvalue pointer to any (optionally cv-qualified) object type T can be converted to a prvalue pointer to (identically cv-qualified) `void`. The resulting pointer represents the same location in memory as the original pointer value. If the original pointer is a null pointer value, the result is a null pointer value of the destination type.
- A prvalue pointer to a (optionally cv-qualified) derived class type can be converted to a prvalue pointer to its (identically cv-qualified) base class. If the base class is inaccessible or ambiguous, the conversion is ill-formed (won't compile). The result of the conversion is a pointer to the base class subobject within the pointed-to object. The null pointer value is converted to the null pointer value of the destination type.

Pointer-to-member conversions

- A *null pointer constant* (see `NULL`) can be converted to any pointer-to-member type, and the result is the null member pointer value of that type. Such conversion (known as *null member pointer conversion*) is allowed to convert to a cv-qualified type as a single conversion, that is, it's not considered a combination of numeric and qualifying conversions.
- A prvalue pointer to member of some type T in a base class B can be converted to a prvalue pointer to member of the same type T in its derived class D. If B is inaccessible, ambiguous, or virtual base of D or is a base of some intermediate virtual base of D, the conversion is ill-formed (won't compile). The resulting pointer can be dereferenced with a D object, and it will access the member within the B base subobject of that D object. The null pointer value is converted to the null pointer value of the destination type.

Boolean conversions

A prvalue of integral, floating-point, unscoped (since C++11) enumeration, pointer, and pointer-to-member types can be converted to a prvalue of type `bool`.

The value zero (for integral, floating-point, and unscoped (since C++11) enumeration) and the null pointer and the null pointer-to-member values become `false`. All other values become `true`.

In the context of a direct-initialization, a `bool` object may be initialized from a prvalue of type `std::nullptr_t`, including `nullptr`. The resulting value is `false`. However, this is not considered to be an implicit conversion. (since C++11)

Qualification conversions

- A prvalue of type pointer to cv-qualified type T can be converted to a prvalue pointer to a more cv-qualified same type T (in other words, constness and volatility can be added).
- A prvalue of type pointer to member of cv-qualified type T in class X can be converted to a prvalue pointer to member of more cv-qualified type T in class X.

"More" cv-qualified means that

- a pointer to *unqualified* type can be converted to a pointer to `const`;
- a pointer to *unqualified* type can be converted to a pointer to `volatile`;
- a pointer to *unqualified* type can be converted to a pointer to `const volatile`;
- a pointer to `const` type can be converted to a pointer to `const volatile`;
- a pointer to `volatile` type can be converted to a pointer to `const volatile`.

For multi-level pointers, the following restrictions apply: a multilevel pointer P1 which is cv_0^1-qualified pointer to cv_1^1-qualified pointer to ... cv_{n-1}^1-qualified pointer to cv_n^1-qualified T is convertible to a multilevel pointer P2 which is cv_0^2-qualified pointer to cv_1^2-qualified pointer to ... cv_{n-1}^2-qualified pointer to cv_n^2-qualified T only if

- the number of levels n is the same for both pointers;
- at every level that array type is involved in, at least one array type has unknown bound, or both array types have same size; (since C++20)
- if there is a `const` in the cv_k^1 qualification at some level (other than level zero) of P1, there is a `const` in the same level cv_k^2 of P2;
- if there is a `volatile` in the cv_k^1 qualification at some level (other than level zero) of P1, there is a `volatile` in the same cv_k^2level of P2;
- if there is an array type of unknown bound at some level (other than level zero) of P1, there is an array type of unknown bound in the same level of P2; (since C++20)
- if at some level k the P2 is *more* cv-qualified than P1 or there is an array type of known bound in P1 and an array type of unknown bound in P2 (since C++20), then there must be a `const` at every single level (other than level zero) of P2 up until k: cv_1^2, cv_2^2, ... cv_k^2.
- same rules apply to multi-level pointers to members and multi-level mixed pointers to objects and pointers to members;
- same rules apply to multi-level pointers that include pointers to array of known or unknown bound at any level (arrays of cv-qualified elements are considered to be identically cv-qualified themselves);
- level zero is addressed by the rules for non-multilevel qualification conversions.

```
char** p = 0;
const char** p1 = p; // error: level 2 more cv-qualified but level 1 is not const
const char* const * p2 = p; // OK: level 2 more cv-qualified and const added at level
volatile char * const * p3 = p; // OK: level 2 more cv-qual and const added at level
volatile const char* const* p4 = p2; // OK: 2 more cv-qual and const was already at 1

double *a[2][3];
double const * const (*ap)[3] = a; // OK
double * const (*ap1)[] = a;       // OK since C++20
```

Note that in the C programming language, const/volatile can be added to the first level only:

```
char** p = 0;
char * const* p1 = p;  // OK in C and C++
const char* const * p2 = p; // error in C, OK in C++
```

Function pointer conversions

- A prvalue of type pointer to non-throwing function can be converted to a prvalue pointer to potentially-throwing function.
- A prvalue of type pointer to non-throwing member function can be converted to a prvalue pointer to potentially-throwing member function.

```
void (*p)();
void (*pp)() noexcept = &p; // error: cannot convert to pointer to noexcept function

struct S
{
    typedef void (*p)();
    operator p();
};
void (*q)() noexcept = S(); // error: cannot convert to pointer to noexcept function
```

(since C++17)

The safe bool problem

Until the introduction of explicit conversion functions in C++11, designing a class that should be usable in boolean contexts (e.g. `if(obj) { ... }`) presented a problem: given a user-defined conversion function, such as `T::operator bool() const;`, the implicit conversion sequence allowed one additional standard conversion sequence after that function call, which means the resultant `bool` could be converted to `int`, allowing such code as `obj << 1;` or `int i = obj;`.

One early solution for this can be seen in `std::basic_ios`, which defines `operator!` and `operator void*` (until C++11), so that the code such as `if(std::cin) {...}` compiles because `void*` is convertible to `bool`, but `int n = std::cout;` does not compile because `void*` is not convertible to `int`. This still allows nonsense code such as `delete std::cout;` to compile, and many pre-C++11 third party libraries were designed with a more elaborate solution, known as the Safe Bool idiom (http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Safe_bool).

The explicit bool conversion can also be used to resolve the safe bool problem

```
explicit operator bool() const { ... }
```

(since C++11)

Footnotes

1. † This only applies if the arithmetic is two's complement which is only required for the exact-width integer types. Note, however, that at the moment all platforms with a C++ compiler use two's complement arithmetic

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 172 (https://cplusplus.github.io/CWG/issues/172.html)	C++98	enumeration type is promoted based on its underlying type	based on its value range instead
CWG 330 (https://cplusplus.github.io/CWG/issues/330.html)	C++98	conversion from `double * const (*p)[3]` to `double const * const (*p)[3]` invalid	conversion valid
CWG 519 (https://cplusplus.github.io/CWG/issues/519.html)	C++98	null pointer values were not guaranteed to be preserved when converting to another pointer type	always preserved
CWG 616 (https://cplusplus.github.io/CWG/issues/616.html)	C++98	the behavior of lvalue to rvalue conversion of any uninitialized object and pointer objects of invalid values was always undefined	indeterminate `unsigned char` is allowed; use of invalid pointers is implementation-defined
CWG 685 (https://cplusplus.github.io/CWG/issues/685.html)	C++98	the underlying type of an enumeration type was not prioritized in integral promotion if it is fixed	prioritized
CWG 707 (https://cplusplus.github.io/CWG/issues/707.html)	C++98	integer to floating point conversion had defined behavior in all cases	the behavior is undefined if the value being converted is out of the destination range
CWG 1423 (https://cplusplus.github.io/CWG/issues/1423.html)	C++11	`std::nullptr_t` is convertible to bool in both direct- and copy-initialization	direct-initialization only
CWG 1781 (https://cplusplus.github.io/CWG/issues/1781.html)	C++11	`std::nullptr_t` to bool is considered a implicit conversion even though it is only valid for direct-initialization	no longer considered an implicit conversion
CWG 1787 (https://cplusplus.github.io/CWG/issues/1787.html)	C++98	the behavior of reading from an indeterminate `unsigned char` cached in a register was undefined	made well-defined
CWG 2484 (https://cplusplus.github.io/CWG/issues/2484.html)	C++20	`char8_t` and `char16_t` have different integral promotion strategies, but they can fit both of them	`char8_t` should be promoted in the same way as `char16_t`

See also

- `const_cast`
- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`
- explicit cast
- user-defined conversion

C documentation for **Implicit conversions**