Implicit conversions

When an expression is used in the context where a value of a different type is expected, *conversion* may occur:

```
int n = 1L; // expression 1L has type long, int is expected
n = 2.1; // expression 2.1 has type double, int is expected
char *p = malloc(10); // expression malloc(10) has type void*, char* is expected
```

Conversions take place in the following situations:

Conversion as if by assignment

- In the assignment operator, the value of the right-hand operand is converted to the unqualified type of the left-hand operand.
- In scalar initialization, the value of the initializer expression is converted to the unqualified type of the object being initialized
- In a function-call expression, to a function that has a prototype, the value of each argument expression is converted to the type of the unqualified declared types of the corresponding parameter
- In a return statement, the value of the operand of `return` is converted to an object having the return type of the function

Note that actual assignment, in addition to the conversion, also removes extra range and precision from floating-point types and prohibits overlaps; those characteristics do not apply to conversion as if by assignment.

Default argument promotions

In a function call expression when the call is made to

1) a function without a prototype
2) a variadic function, where the argument expression is one of the trailing arguments that are matched against the ellipsis parameter

Each argument of integer type undergoes *integer promotion* (see below), and each argument of type `float` is implicitly converted to the type `double`

```
int add_nums(int count, ...);
int sum = add_nums(2, 'c', true); // add_nums is called with three ints: (2, 99, 1)
```

Note that `float complex` and `float imaginary` are not promoted to `double complex` and `double imaginary` in this context.

Usual arithmetic conversions

The arguments of the following arithmetic operators undergo implicit conversions for the purpose of obtaining the *common real type*, which is the type in which the calculation is performed:

- binary arithmetic *, /, %, +, -
- relational operators <, >, <=, >=, ==, !=
- binary bitwise arithmetic &, ^, |,
- the conditional operator ?:

1) If one operand is `long double`, `long double complex`, or `long double imaginary`, the other operand is implicitly converted as follows:
    - integer or real floating type to `long double`
    - complex type to `long double complex`
    - imaginary type to `long double imaginary`
2) Otherwise, if one operand is `double`, `double complex`, or `double imaginary`, the other operand is implicitly converted as follows:
    - integer or real floating type to `double`
    - complex type to `double complex`
    - imaginary type to `double imaginary`
3) Otherwise, if one operand is `float`, `float complex`, or `float imaginary`, the other operand is implicitly converted as follows:

- integer type to `float` (the only real type possible is float, which remains as-is)
- complex type remains `float complex`
- imaginary type remains `float imaginary`

4) Otherwise, both operands are integers. In that case,

First of all, both operands undergo *integer promotions* (see below). Then

- If the types after promotion are the same, that type is the common type
- Otherwise, if both operands after promotion have the same signedness (both signed or both unsigned), the operand with the lesser *conversion rank* (see below) is implicitly converted to the type of the operand with the greater *conversion rank*
- Otherwise, the signedness is different: If the operand with the unsigned type has *conversion rank* greater or equal than the rank of the type of the signed operand, then the operand with the signed type is implicitly converted to the unsigned type
- Otherwise, the signedness is different and the signed operand's rank is greater than unsigned operand's rank. In this case, if the signed type can represent all values of the unsigned type, then the operand with the unsigned type is implicitly converted to the type of the signed operand.
- Otherwise, both operands undergo implicit conversion to the unsigned type counterpart of the signed operand's type.

```
1.f + 20000001; // int is converted to float, giving 20000000.00
                // addition and then rounding to float gives 20000000.00
(char)'a' + 1L; // First, char is promoted back to int.
                // this is signed + signed case, different rank
                // int is converted to long, the result is 98 signed long
2u - 10; // signed / unsigned, same rank
         // 10 is converted to unsigned, unsigned math is modulo UINT_MAX+1
         // for 32 bit ints, result is 4294967288 of type unsigned int (aka UINT_MAX-7)
0UL - 1LL; // signed/unsigned diff rank, rank of signed is greater.
           // If sizeof(long) == sizeof(long long), signed cannot represent all unsigned
           // this is the last case: both operands are converted to unsigned long long
           // the result is 18446744073709551615 (ULLONG_MAX) of type unsigned long long
```

The result type is determined as follows:

- if both operands are complex, the result type is complex
- if both operands are imaginary, the result type is imaginary
- if both operands are real, the result type is real
- if the two floating-point operands have different type domains (complex vs. real, complex vs imaginary, or imaginary vs. real), the result type is complex

```
double complex z = 1 + 2*I;
double f = 3.0;
z + f; // z remains as-is, f is converted to double, the result is double complex
```

As always, the result of a floating-point operator may have greater range and precision than is indicated by its type (see `FLT_EVAL_METHOD`).

Note: real and imaginary operands are not implicitly converted to complex because doing so would require extra computation, while producing undesirable results in certain cases involving infinities, NaNs and signed zeros. For example, if reals were converted to complex, 2.0×(3.0+i∞) would evaluate as (2.0+i0.0)×(3.0+i∞) ⇒ (2.0×3.0–0.0×∞) + i(2.0×∞+0.0×3.0) ⇒ NaN+i∞ rather than the correct 6.0+i∞. If imaginaries were converted to complex, i2.0×(∞+i3.0) would evaluate as (0.0+i2.0) × (∞+i3.0) ⇒ (0.0×∞ – 2.0×3.0) + i(0.0×3.0 + 2.0×∞) ⇒ NaN + i∞ instead of –6.0 + i∞.

Note: regardless of usual arithmetic conversions, the calculation may always be performed in a narrower type than specifier by these rules under the as-if rule

Value transformations

Lvalue conversion

Any lvalue expression of any non-array type, when used in any context other than

- as the operand of the address-of operator (if allowed)
- as the operand of the pre/post increment and decrement operators.
- as the left-hand operand of the member access (dot) operator.
- as the left-hand operand of the assignment and compound assignment operators.
- as the operand of sizeof

undergoes *lvalue conversion*: the type remains the same, but loses const/volatile/restrict-qualifiers and atomic properties, if any. The value remains the same, but loses its lvalue properties (the address may no longer be taken).

If the lvalue has incomplete type, the behavior is undefined.

If the lvalue designates an object of automatic storage duration whose address was never taken and if that object was uninitialized (not declared with an initializer and no assignment to it has been performed prior to use), the behavior is undefined.

This conversion models the memory load of the value of the object from its location.

```
volatile int n = 1;
int x = n;            // lvalue conversion on n reads the value of n
volatile int* p = &n; // no lvalue conversion: does not read the value of n
```

Array to pointer conversion

Any lvalue expression of array type, when used in any context other than

- as the operand of the address-of operator
- as the operand of sizeof
- as the string literal used for array initialization

undergoes a conversion to the non-lvalue pointer to its first element.

If the array was declared register, the behavior is undefined.

```
int a[3], b[3][4];
int* p = a;      // conversion to &a[0]
int (*q)[4] = b; // conversion to &b[0]
```

Function to pointer conversion

Any function designator expression, when used in any context other than

- as the operand of the address-of operator
- as the operand of sizeof

undergoes a conversion to the non-lvalue pointer to the function designated by the expression.

```
int f(int);
int (*p)(int) = f; // conversion to &f
(***p)(1); // repeated dereference to f and conversion back to &f
```

Implicit conversion semantics

Implicit conversion, whether *as if by assignment* or a *usual arithmetic conversion*, consists of two stages:

1) value transformation (if applicable)
2) one of the conversions listed below (if it can produce the target type)

Compatible types

Conversion of a value of any type to any compatible type is always a no-op and does not change the representation.

```
uint8_t (*a)[10];         // if uint8_t is a typedef to unsigned char
unsigned char (*b)[] = a; // then these pointer types are compatible
```

Integer promotions

Integer promotion is the implicit conversion of a value of any integer type with *rank* less or equal to *rank* of int or of a bit field of type _Bool, int, signed int, unsigned int, to the value of type `int` or `unsigned int`

If `int` can represent the entire range of values of the original type (or the range of values of the original bit field), the value is converted to type `int`. Otherwise the value is converted to `unsigned int`.

Integer promotions preserve the value, including the sign:

```
int main(void) {
   void f(); // old-style function declaration
```

```
    char x = 'a'; // integer conversion from int to char
    f(x); // integer promotion from char back to int
}
void f(x) int x; {} // the function expects int
```

rank above is a property of every integer type and is defined as follows:

1) the ranks of all signed integer types are different and increase with their precision: rank of signed char < rank of short < rank of int < rank of long int < rank of long long int
2) the ranks of all signed integer types equal the ranks of the corresponding unsigned integer types
3) the rank of any standard integer type is greater than the rank of any extended integer type of the same size (that is, rank of __int64 < rank of long long int, but rank of long long < rank of __int128 due to the rule (1))
4) rank of char equals rank of signed char and rank of unsigned char
5) the rank of _Bool is less than the rank of any other standard integer type
6) the rank of any enumerated type equals the rank of its compatible integer type
7) ranking is transitive: if rank of T1 < rank of T2 and rank of T2 < rank of T3 then rank of T1 < rank of T3
8) any aspects of relative ranking of extended integer types not covered above are implementation defined

Note: integer promotions are applied only

- as part of *usual arithmetic conversions* (see above)
- as part of *default argument promotions* (see above)
- to the operand of the unary arithmetic operators + and -
- to the operand of the unary bitwise operator ~
- to both operands of the shift operators << and >>

Boolean conversion

A value of any scalar type can be implicitly converted to _Bool. The values that compare equal to zero are converted to `0`, all other values are converted to `1`

```
bool b1 = 0.5;              // b1 == 1 (0.5 converted to int would be zero)
bool b2 = 2.0*_Imaginary_I; // b2 == 1 (but converted to int would be zero)
bool b3 = 0.0 + 3.0*I;      // b3 == 1 (but converted to int would be zero)
bool b4 = 0.0/0.0;          // b4 == 1 (NaN does not compare equal to zero)
```

Integer conversions

A value of any integer type can be implicitly converted to any other integer type. Except where covered by promotions and boolean conversions above, the rules are:

- if the target type can represent the value, the value is unchanged
- otherwise, if the target type is unsigned, the value 2^b, where b is the number of bits in the target type, is repeatedly subtracted or added to the source value until the result fits in the target type. In other words, unsigned integers implement modulo arithmetic.
- otherwise, if the target type is signed, the behavior is implementation-defined (which may include raising a signal)

```
char x = 'a'; // int -> char, result unchanged
unsigned char n = -123456; // target is unsigned, result is 192 (that is, -123456+483*256)
signed char m = 123456;    // target is signed, result is implementation-defined
assert(sizeof(int) > -1);  // assert fails:
                           // operator > requests conversion of -1 to size_t,
                           // target is unsigned, result is SIZE_MAX
```

Real floating-integer conversions

A finite value of any real floating type can be implicitly converted to any integer type. Except where covered by boolean conversion above, the rules are:

- The fractional part is discarded (truncated towards zero).
    - If the resulting value can be represented by the target type, that value is used
    - otherwise, the behavior is undefined

```
int n = 3.14; // n == 3
int x = 1e10; // undefined behavior for 32-bit int
```

A value of any integer type can be implicitly converted to any real floating type.

- if the value can be represented exactly by the target type, it is unchanged
- if the value can be represented, but cannot be represented exactly, the result is the nearest higher or the nearest lower value (in other words, rounding direction is implementation-defined), although if IEEE arithmetic is supported, rounding is to nearest. It is unspecified whether `FE_INEXACT` is raised in this case.
- if the value cannot be represented, the behavior is undefined, although if IEEE arithmetic is supported, `FE_INVALID` is raised and the result value is unspecified.

The result of this conversion may have greater range and precision than its target type indicates (see `FLT_EVAL_METHOD`.

If control over `FE_INEXACT` is needed in floating-to-integer conversions, `rint` and `nearbyint` may be used.

```
double d = 10; // d = 10.00
float f = 20000001; // f = 20000000.00 (FE_INEXACT)
float x = 1+(long long)FLT_MAX; // undefined behavior
```

Real floating point conversions

A value of any real floating type can be implicitly converted to any other real floating type.

- If the value can be represented by the target type exactly, it is unchanged
- if the value can be represented, but cannot be represented exactly, the result is the nearest higher or the nearest lower value (in other words, rounding direction is implementation-defined), although if IEEE arithmetic is supported, rounding is to nearest
- if the value cannot be represented, the behavior is undefined

> This section is incomplete
> Reason: check IEEE if appropriately-signed infinity is required

The result of this conversion may have greater range and precision than its target type indicates (see `FLT_EVAL_METHOD`.

```
double d = 0.1; // d = 0.1000000000000000055511151231257827021181583404541015625
float f = d;    // f = 0.100000001490116119384765625
float x = 2*(double)FLT_MAX; // undefined
```

Complex type conversions

A value of any complex type can be implicitly converted to any other complex type. The real part and the imaginary part individually follow the conversion rules for the real floating types.

```
double complex d = 0.1 + 0.1*I;
float complex f = d; // f is (0.100000001490116119384765625, 0.100000001490116119384765625)
```

Imaginary type conversions

A value of any imaginary type can be implicitly converted to any other imaginary type. The imaginary part follows the conversion rules for the real floating types.

```
double imaginary d = 0.1*_Imaginary_I;
float imaginary f = d; // f is 0.100000001490116119384765625*I
```

Real-complex conversions

A value of any real floating type can be implicitly converted to any complex type.

- The real part of the result is determined by the conversion rules for the real floating types
- The imaginary part of the result is positive zero (or unsigned zero on non-IEEE systems)

A value of any complex type can be implicitly converted to any real floating type

- The real part is converted following the rules for the real floating types
- The imaginary part is discarded

Note: in complex-to-real conversion, a NaN in the imaginary part will not propagate to the real result.

```
double complex z = 0.5 + 3*I;
float f = z;  // the imaginary part is discarded, f is set to 0.5
z = f;        // sets z to 0.5 + 0*I
```

Real-imaginary conversions

A value of any imaginary type can be implicitly converted to any real type (integer or floating-point). The result is always a positive (or unsigned) zero, except when the target type is _Bool, in which case boolean conversion rules apply.

A value of any real type can be implicitly converted to any imaginary type. The result is always a positive imaginary zero.

```
double imaginary z = 3*I;
bool b = z;  // Boolean conversion: sets b to true
float f = z; // Real-imaginary conversion: sets f to 0.0
z = 3.14;    // Imaginary-real conversion: sets z to 0*_Imaginary_I
```

Complex-imaginary conversions

A value of any imaginary type can be implicitly converted to any complex type.

- The real part of the result is the positive zero
- The imaginary part of the result follows the conversion rules for the corresponding real types

A value of any complex type can be implicitly converted to any imaginary type

- The real part is discarded
- The imaginary part of the result follows the conversion rules for the corresponding real types

```
double imaginary z = I * (3*I); // the complex result -3.0+0i loses real part
                                // sets z to 0*_Imaginary_I
```

Pointer conversions

A pointer to `void` can be implicitly converted to and from any pointer to object type with the following semantics:

- If a pointer to object is converted to a pointer to void and back, its value compares equal to the original pointer.
- No other guarantees are offered

```
int* p = malloc(10 * sizeof(int)); // malloc returns void*
```

A pointer to an unqualified type may be implicitly converted to the pointer to qualified version of that type (in other words, const, volatile, and restrict qualifiers can be added. The original pointer and the result compare equal.

```
int n;
const int* p = &n; // &n has type int*
```

Any integer constant expression with value `0` as well as integer pointer expression with value zero cast to the type `void*` can be implicitly converted to any pointer type (both pointer to object and pointer to function). The result is the null pointer value of its type, guaranteed to compare unequal to any non-null pointer value of that type. This integer or void* expression is known as *null pointer constant* and the standard library provides one definition of this constant as the macro `NULL` .

```
int* p = 0;
double* q = NULL;
```

Notes

Although signed integer overflow in any arithmetic operator is undefined behavior, overflowing a signed integer type in an integer conversion is merely unspecified behavior.

On the other hand, although unsigned integer overflow in any arithmetic operator (and in integer conversion) is a well-defined operation and follows the rules of modulo arithmetic, overflowing an unsigned integer in a floating-to-integer conversion is undefined behavior: the values of real floating type that can be converted to unsigned integer are the values from the open interval $(-1; Unnn\_MAX+1)$.

```
unsigned int n = -1.0; // undefined behavior
```

Conversions between pointers and integers (except from pointer to _Bool and from integer constant expression with the value zero to pointer), between pointers to objects (except where either to or from is a pointer to void) and conversions between pointers to functions (except when the functions have compatible types) are never implicit and require a cast operator.

There are no conversions (implicit or explicit) between pointers to functions and pointers to objects (including void*) or integers.

References

- C11 standard (ISO/IEC 9899:2011):
  - 6.3 Conversions (p: 50-56)
- C99 standard (ISO/IEC 9899:1999):
  - 6.3 Conversions (p: 42-48)
- C89/C90 standard (ISO/IEC 9899:1990):
  - 3.2 Conversions

See also

C++ documentation for **`Implicit conversions`**