Implicit type promotion rules

Asked 6 years ago Modified 8 days ago

Ask Question

134

This post is meant to be used as a FAQ regarding implicit integer promotion in C, particularly implicit promotion caused by the usual arithmetic conversions and/or the integer promotions.

Example 1)
Why does this give a strange, large integer number and not 255?

```
unsigned char x = 0;
unsigned char y = 1;
printf("%u\n", x - y);
```

Example 2)
Why does this give "-1 is larger than 0"?

```
unsigned int a = 1;
signed int b = -2;
if(a + b > 0)
  puts("-1 is larger than 0");
```

Example 3)
Why does changing the type in the above example to `short` fix the problem?

```
unsigned short a = 1;
signed short b = -2;
if(a + b > 0)
  puts("-1 is larger than 0"); // will not print
```

(These examples were intended for a 32 or 64 bit computer with 16 bit short.)

c type-conversion implicit-conversion

Share Improve this question Follow edited Jun 17, 2022 at 5:58 asked Sep 6, 2017 at 10:50 Lundin 196k 40 254 398

4 I suggest documenting the assumptions for the examples, e.g. example 3 assumes that `short` is narrower than `int` (or in other words, it assumes that `int` can represent all the values of `unsigned short`). – Ian Abbott Sep 6, 2017 at 12:41

6 @savram Yes, the intention is to write a FAQ entry. Sharing knowledge this way is fine for SO - next time you post a question note the checkbox "answer your own question". But of course the question is still treated like any other question and others can post answers too. (And you don't earn any rep from accepting your own answer) – Lundin Sep 7, 2017 at 6:31

2 @savram: It is absolutely fine to share knowledge this way. See here: self answer. – Andre Kampling Sep 7, 2017 at 12:06

3 Neither answer so far mentions the fact that `printf("%u\n", x - y);` causes undefined behaviour – M.M Aug 20, 2019 at 21:45

2 Nice example is `~((u8)(1 << 7))` to the list. – 0andriy Feb 25, 2020 at 14:38

Show 3 more comments

5 Answers

Sorted by: Highest score (default)

187

C was designed to implicitly and silently change the integer types of the operands used in expressions. There exist several cases where the language forces the compiler to either change the operands to a larger type, or to change their signedness.

The rationale behind this is to prevent accidental overflows during arithmetic, but also to allow operands with different signedness to co-exist in the same expression.

Unfortunately, the rules for implicit type promotion cause much more harm than good, to the point where they might be one of the biggest flaws in the C language. These rules are often not even known by the average C programmer and therefore cause all manner of very subtle bugs.

Typically you see scenarios where the programmer says "just cast to type x and it works" - but they don't know why. Or such bugs manifest themselves as rare, intermittent phenomena striking from within seemingly simple and straight-forward code. Implicit promotion is particularly troublesome in code doing bit manipulations, since most bit-wise operators in C come with poorly-defined behavior when given a signed operand.

Integer types and conversion rank

The integer types in C are `char`, `short`, `int`, `long`, `long long` and `enum`.
`_Bool`/`bool` is also treated as an integer type when it comes to type promotions.

All integers have a specified *conversion rank*. C11 6.3.1.1, emphasis mine on the most important parts:

> Every integer type has an integer conversion rank defined as follows:
> — No two signed integer types shall have the same rank, even if they have the same representation.
> — The rank of a signed integer type shall be greater than the rank of any signed integer type with less precision.
> — **The rank of `long long int` shall be greater than the rank of `long int`, which shall be greater than the rank of `int`, which shall be greater than the rank of `short int`, which shall be greater than the rank of `signed char`.**
> — **The rank of any unsigned integer type shall equal the rank of the corresponding signed integer type, if any.**
> — The rank of any standard integer type shall be greater than the rank of any extended integer type with the same width.
> — The rank of char shall equal the rank of signed char and unsigned char.
> — The rank of _Bool shall be less than the rank of all other standard integer types.
> — The rank of any enumerated type shall equal the rank of the compatible integer type (see 6.7.2.2).

The types from `stdint.h` sort in here too, with the same rank as whatever type they happen to correspond to on the given system. For example, `int32_t` has the same rank as `int` on a 32 bit system.

Further, C11 6.3.1.1 specifies which types are regarded as the *small integer types* (not a formal term):

> The following may be used in an expression wherever an `int` or `unsigned int` may be used:
>
> — An object or expression with an integer type (other than `int` or `unsigned int`) whose integer conversion rank is less than or equal to the rank of `int` and `unsigned int`.

What this somewhat cryptic text means in practice, is that `_Bool`, `char` and `short` (and also `int8_t`, `uint8_t` etc) are the "small integer types". These are treated in special ways and subject to implicit promotion, as explained below.

The integer promotions

Whenever a small integer type is used in an expression, it is implicitly converted to `int` which is always signed. This is known as the *integer promotions* or *the integer promotion rule*.

Formally, the rule says (C11 6.3.1.1):

> If an `int` can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called the *integer promotions*.

This means that all small integer types, no matter signedness, get implicitly converted to (signed) `int` when used in most expressions.

This text is often misunderstood as: "all small signed integer types are converted to signed int and all small, unsigned integer types are converted to unsigned int". This is incorrect. The unsigned part here only means that if we have for example an `unsigned short` operand, and `int` happens to have the same size as `short` on the given system, then the `unsigned short` operand is converted to `unsigned int`. As in, nothing of note really happens. But in case `short` is a smaller type than `int`, it is always converted to (signed) `int`, *regardless of it the short was signed or unsigned*!

The harsh reality caused by the integer promotions means that almost no operation in C can be carried out on small types like `char` or `short`. Operations are always carried out on `int` or larger types.

This might sound like nonsense, but luckily the compiler is allowed to optimize the code. For example, an expression containing two `unsigned char` operands would get the operands promoted to `int` and the operation carried out as `int`. But the compiler is allowed to optimize the expression to actually get carried out as an 8-bit operation, as would be expected. However, here comes the problem: the compiler is *not* allowed to optimize out the implicit change of signedness caused by the integer promotion because there is no way for the compiler to tell if the programmer is purposely relying on implicit promotion to happen, or if it is unintentional.

This is why example 1 in the question fails. Both unsigned char operands are promoted to type `int`, the operation is carried out on type `int`, and the result of `x - y` is of type `int`. Meaning that we get `-1` instead of `255` which might have been expected. The compiler may generate machine code that executes the code with 8 bit instructions instead of `int`, but it may not optimize out the change of signedness. Meaning that we end up with a negative result, that in turn results in a weird number when `printf("%u` is invoked. Example 1 could be fixed by casting the result of the operation back to type `unsigned char`.

With the exception of a few special cases like `++` and `sizeof` operators, the integer promotions apply to almost all operations in C, no matter if unary, binary (or ternary) operators are used.

The usual arithmetic conversions

Whenever a binary operation (an operation with 2 operands) is done in C, both operands of the operator have to be of the same type. Therefore, in case the operands are of different types, C enforces an implicit conversion of one operand to the type of the other operand. The rules for how this is done are named *the usual artihmetic conversions* (sometimes informally referred to as "balancing"). These are specified in C11 6.3.18:

(Think of this rule as a long, nested `if-else if` statement and it might be easier to read :))

> 6.3.1.8 Usual arithmetic conversions
>
> Many operators that expect operands of arithmetic type cause conversions and yield result types in a similar way. The purpose is to determine a common real type for the operands and result. For the specified operands, each operand is converted, without change of type domain, to a type whose corresponding real type is the common real type. Unless explicitly stated otherwise, the common real type is also the corresponding real type of the result, whose type domain is the type domain of the operands if they are the same, and complex otherwise. This pattern is called *the usual arithmetic conversions*:
>
> - First, if the corresponding real type of either operand is `long double`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `long double`.
> - Otherwise, if the corresponding real type of either operand is `double`, the other operand is converted, without change of type domain, to a type whose corresponding real type is `double`.
> - Otherwise, if the corresponding real type of either operand is `float`, the other operand is converted, without change of type domain, to a type whose corresponding real type is float.
> - Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
>   - If both operands have the same type, then no further conversion is needed.
> - Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
> - Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
> - Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
> - Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Notable here is that the usual arithmetic conversions apply to both floating point and integer variables. In the case of integers, we can also note that the integer promotions are invoked from within the usual arithmetic conversions. And after that, when both operands have at least the rank of `int`, the operators are balanced to the same type, with the same signedness.

This is the reason why `a + b` in example 2 gives a strange result. Both operands are integers and they are at least of rank `int`, so the integer promotions do not apply. The operands are not of the same type - `a` is `unsigned int` and `b` is `signed int`. Therefore the operator `b` is temporarily converted to type `unsigned int`. During this conversion, it loses the sign information and ends up as a large value.

The reason why changing type to `short` in example 3 fixes the problem, is because `short` is a small integer type. Meaning that both operands are integer promoted to type `int` which is signed. After integer promotion, both operands have the same type (`int`), no further conversion is needed. And then the operation can be carried out on a signed type as expected.

Of note, C++ applies pretty much identical rules.

Share Improve this answer Follow edited Sep 14 at 22:50 Martin Dorey 2,944 2 24 16 answered Sep 6, 2017 at 10:50 Lundin 196k 40 254 398

1 *"Whenever a small integer type is used in an expression, it is implicitly converted to int which is always signed."* Could you point to the exact place in the standard that says that it *should* happen? The C11 6.3.1.1 quote says *how* it happens (if it happens) but it doesn't say that it *must* happen e.g., why `x - y` in the question behaves as `(unsigned)(int)((int)x - (int)y)` instead of `(unsigned)(int)((Uchar)((Uchar)x - (Uchar)y))` goo.gl/nCvJy5 . Where does the standard says that if `x` is char then `+x` is `int` (or unsigned)? In c++ it is §5.3.1.7 goo.gl/FkEakX – jfs Nov 17, 2017 at 11:31

2 @jfs "Otherwise, ..." (if neither operand is float type) "...the integer promotions are performed on both operands." . And then afterwards "If both operands have the same type, then no further conversion is needed." – Lundin Nov 17, 2017 at 14:14

1 "Example 1 could be fixed by casting one or both operands to type unsigned int." The suggested cast(s) won't yield 255 as the OP expected. The proper fix is to cast the result of the subtraction back to the `(unsigned char)` that the operands started from, as in `(unsigned char) (x-y)`: this will give the OP the expected 255. People often fail to appreciate casting to a smaller size, however, this is the proper way to accomplish truncation (which will be followed by implicit/automatic signed or zero extension to ~int size). – Erik Eidt Jan 6, 2019 at 18:32

1 @Lundin The types defined by stdint.h may be extended integer types. See C17/C18 footnote 265, C11 footnote 261, or C99 footnote 218: **"Some of these types may denote implementation-defined extended integer types."** Any such types would have lesser rank than a standard integer type of the same width. (An example that springs to mind - if the standard signed types are ones-complement, but there is a special twos-complement keyword to define `int32_t`, etc.) – Ian Abbott Sep 2, 2021 at 13:11

1 @JasonS It's said in normative text for each operator where it applies, for example 6.5.7 "The integer promotions are performed on each of the operands." or 6.5.5 "The usual arithmetic conversions are performed on the operands." – Lundin Oct 31, 2022 at 7:08

Show 15 more comments

10

According to the previous post, I want to give more information about each example.

Example 1)

```
int main(){
    unsigned char x = 0;
    unsigned char y = 1;
    printf("%u\n", x - y);
    printf("%d\n", x - y);
}
```

Since unsigned char is smaller than int, we apply the integer promotion on them, then we have (int)x-(int)y = (int)(-1) and unsigned int (-1) = 4294967295.

The output from the above code:(same as what we expected)

```
4294967295
-1
```

How to fix it?

I tried what the previous post recommended, but it doesn't really work. Here is the code based on the previous post:

change one of them to unsigned int

```
int main(){
    unsigned int x = 0;
    unsigned char y = 1;
    printf("%u\n", x - y);
    printf("%d\n", x - y);
}
```

Since x is already an unsigned integer, we only apply the integer promotion to y. Then we get (unsigned int)x-(int)y. Since they still don't have the same type, we apply the usual arithmetic converions, we get (unsigned int)x-(unsigned int)y = 4294967295.

The output from the above code:(same as what we expected):

```
4294967295
-1
```

Similarly, the following code gets the same result:

```
int main(){
    unsigned char x = 0;
    unsigned int y = 1;
    printf("%u\n", x - y);
    printf("%d\n", x - y);
}
```

change both of them to unsigned int

```
int main(){
    unsigned int x = 0;
    unsigned int y = 1;
    printf("%u\n", x - y);
    printf("%d\n", x - y);
}
```

Since both of them are unsigned int, no integer promotion is needed. By the usual arithmetic converison(have the same type), (unsigned int)x-(unsigned int)y = 4294967295.

The output from the above code:(same as what we expected):

```
4294967295
-1
```

One of possible ways to fix the code:(add a type cast in the end)

```
int main(){
    unsigned char x = 0;
    unsigned char y = 1;
    printf("%u\n", x - y);
    printf("%d\n", x - y);
    unsigned char z = x-y;
    printf("%u\n", z);
}
```

The output from the above code:

```
4294967295
-1
255
```

Example 2)

```
int main(){
    unsigned int a = 1;
    signed int b = -2;
    if(a + b > 0)
        puts("-1 is larger than 0");
        printf("%u\n", a+b);
}
```

Since both of them are integers, no integer promotion is needed. By the usual arithmetic conversion, we get (unsigned int)a+(unsigned int)b = 1+4294967294 = 4294967295.

The output from the above code:(same as what we expected)

```
-1 is larger than 0
4294967295
```

How to fix it?

```
int main(){
    unsigned int a = 1;
    signed int b = -2;
    signed int c = a+b;
    if(c < 0)
        puts("-1 is smaller than 0");
        printf("%d\n", c);
}
```

The output from the above code:

```
-1 is smaller than 0
-1
```

Example 3)

```
int main(){
    unsigned short a = 1;
    signed short b = -2;
    if(a + b < 0)
        puts("-1 is smaller than 0");
        printf("%d\n", a+b);
}
```

The last example fixed the problem since a and b both converted to int due to the integer promotion.

The output from the above code:

```
-1 is smaller than 0
-1
```

If I got some concepts mixed up, please let me know. Thanks~

Share Improve this answer Follow edited Jun 17, 2022 at 3:55 Gabriel Staples 37.1k 15 186 268 answered Jun 29, 2018 at 15:46 Lusha Li 319 3 11

2 Your fix to Example 2 `signed int c = a+b;` above invokes UB. The resultant type of a+b is unsigned, and the computed value is out of range of a signed integer. – Cheshar Jun 5, 2019 at 16:32

2 @Cheshar out-of-range assignment is not UB – M.M Aug 20, 2019 at 21:43

2 many of the examples in this answer cause UB by using the wrong format specifier, and also it makes an unwarranted assumption about the size of an `int` – M.M Aug 20, 2019 at 21:47

1 @M.M My bad! Agree, it should have been "implementation-defined or an implementation-defined signal raised". Signed overflow is UB though. It's easier to lose track of UB/IB. – Cheshar Aug 22, 2019 at 17:26

1 @Cheshar: Contrary to myth spread by some compiler maintainers, the Standard's term for actions which should be processed identically by 99.9% of implementations, but which need not be processed meaningfully by implementations where that would be impractical, is "Undefined Behavior". The term IDB is only used for actions which *all* implementations are supposed to process meaningfully. – supercat Sep 7, 2019 at 22:30

Show 1 more comment

3

Integer and floating point rank and promotion rules in C and C++

I'd like to take a stab at this to summarize the rules so I can quickly reference them. I've fully studied the question and both of the other two answers here, including the main one by @Lundin. If you want more examples beyond the ones below, go study that answer in detail as well, while referencing my "rules" and "promotion flow" summaries below.

I've also written my own example and demo code here: integer_promotion_overflow_underflow_undefined_behavior.c.

Despite normally being incredibly verbose myself, I'm going to try to keep this a short summary, since the other two answers plus my test code already have sufficient detail via their necessary verbosity.

Integer and variable promotion quick reference guide and summary

3 simple rules

1. For any operation where multiple operands (input variables) are involved (ex: mathematical operations, comparisons, or ternary), the variables are *automatically implicitly promoted* as required to the required variable type *before* the operation is performed.
   1. Therefore, you must manually, explicitly *cast the output* to any desired type you desire if you do not want it to be implicitly chosen for you. See the example below.
2. All types smaller than `int` (`int32_t` on my 64-bit Linux system) are "small types". They cannot be used in ANY operation. So, if all input variables are "small types", they are ALL first automatically implicitly promoted to `int` (`int32_t` on my 64-bit Linux system) before performing the operation.
3. Otherwise, if at least one of the input types is `int` or larger, the other, smaller input type or types are automatically implicitly promoted to this largest-input-type's type.

Example

Example: with this code:

```
uint8_t x = 0;
uint8_t y = 1;
```

...if you do `x - y`, they first get implicitly promoted to `int` (which is `int32_t` on my 64-bit system), and you end up with this: `(int)x - (int)y`, which results in an `int` type with value `-1`, rather than a `uint8_t` type of value `255`. To get the desired `255` result, **manually** cast the result back to `uint8_t`, by doing this: `(uint8_t)(x - y)`.

Promotion flow

The promotion rules are as follows. Promotion from *smallest to largest* types is as follows.
Read "`-->`" as "gets promoted to".

The types in square brackets (ex: `[int8_t]`) are the typical "fixed-width integer types" for the given standard type on a typical 64-bit Unix (Linux or Mac) architecture. See, for example:

1. https://www.cs.yale.edu/homes/aspnes/pinewiki/C(2f)IntegerTypes.html
2. https://www.ibm.com/docs/en/ibm-mq/7.5?topic=platforms-standard-data-types
3. And **even better, test it for yourself on your machine** by running my code here!: **stdint_sizes.c** from my eRCaGuy_hello_world repo.

1. For integer types in 64-bit x86-64 architecture CPUs

Note: "*small types*" = `bool` (`_Bool`), `char [int8_t]`, `unsigned char [uint8_t]`, `short [int16_t]`, `unsigned short [uint16_t]`.

SMALL TYPES: `bool` (`_Bool`), `char [int8_t]`, `unsigned char [uint8_t]`, `short [int16_t]`, `unsigned short [uint16_t]`
--> `int [int32_t]`
--> `unsigned int [uint32_t]`
--> `long int [int64_t]`
--> `unsigned long int [uint64_t]`
--> `long long int [int64_t]`
--> `unsigned long long int [uint64_t]`

Pointers (ex: `void*`) and `size_t` are both 64-bits, so I imagine they fit into the `uint64_t` category above.

2. For floating point types

`float [32-bits]` --> `double [64-bits]` --> `long double [128-bits]`

See also

1. https://cppinsights.io/ - a very useful tool which expands your C++ code into exactly what the compiler sees, including after applying all automatic implicit type promotion rules in the compiler.
   1. Ex: see my code from my answer here in CPPInsights.io here: https://cppinsights.io/s/bfc425f6 --> then click the play button to convert and expand it into what the compiler sees, including after applying all automatic implicit type promotion rules.

Places where I use these rules

1. How to safely and efficiently find `abs((int)num1 - (int)num2)`

Share Improve this answer Follow edited Aug 30 at 15:28 answered Jun 17, 2022 at 5:25 Gabriel Staples 37.1k 15 196 268

Add a comment

1

I would like to add two clarifications to @Lundin's otherwise excellent answer, regarding example 1, where there are two operands of identical integer type, but are "small types" that require integer promotion.

I'm using the N1256 draft since I don't have access to a paid copy of the C standard.

First: (normative)

6.3.1.1's definition of integer promotion isn't the triggering clause of actually *doing* integer promotion. In reality it is 6.3.1.8 Usual arithmetic conversions.

Most of the time, the "usual arithmetic conversions" apply when the operands are of *different* types, in which case at least one operand must be promoted. But the catch is that for integer types, integer promotion is required in all cases.

> [clauses of floating-point types come first]
>
> Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
>
> - If both operands have the same type, then no further conversion is needed.
> - Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
> - Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
> - Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
> - Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Second: (non-normative)

There is an explicit example cited by the standard to demonstrate this:

> EXAMPLE 2 In executing the fragment
>
> ```
> char c1, c2;
> /* ... */
> c1 = c1 + c2;
> ```
>
> the "integer promotions" require that the abstract machine promote the value of each variable to `int` size and then add the two `int`s and truncate the sum. Provided the addition of two `char`s can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

10 EXAMPLE 2 In executing the fragment

```
char c1, c2;
/* ... */
c1 = c1 + c2;
```

the "integer promotions" require that the abstract machine promote the value of each variable to **int** size and then add the two **int**s and truncate the sum. Provided the addition of two **char**s can be done without overflow, or with overflow wrapping silently to produce the correct result, the actual execution need only produce the same result, possibly omitting the promotions.

Share Improve this answer Follow answered Oct 30, 2022 at 4:48 Jason S 185k 164 610 971

Add a comment

1

In this answer I'll address the compiler flags you can use to track down bugs related to implicit type promotion since I just ran into this "feature". In the following buggy code fragment `exp` is of type `uint32_t`:

```
for (int32_t i = 22; i >= MAX(22 - exp + 1, 0); i--) {
    ...
}
```

If `exp` < 23 code works fine, if `exp` = 23 loop runs forever, and if `exp` > 23 loop never runs. The fix is to change the first argument to `MAX` to `22 - (int32_t)exp + 1`. To make it easier to spot such bugs I recommend turning on the warning `-Wsign-compare`. It is included in `-Wextra`, which may be a little heavy for everyday use.

The bug in the other example;

```
unsigned short a = 1;
signed short b = -2;
if(a + b > 0)
    puts("-1 is larger than 0"); // will not print
```

is caught by `-Wsign-conversion`, also included in `-Wextra`. In my own codebase this flag produces about 40 warnings all of which are completely benign and not worth the bother to fix.

Unfortunately, neither gcc nor clang has warnings for flagging "suspicious" type promotions, but leaving safe ones be (e.g `for (int i = 0; i < strlen(s); i++)`).

You may want to read Friends don't let friends use "-W" for (informed) opinion on when and when not to use the compiler's warning flags.

Share Improve this answer Follow answered Jul 19 at 14:37 Björn Lindqvist 19.2k 20 87 122

Add a comment

Your Answer

Not the answer you're looking for? Browse other questions tagged c type-conversion implicit-conversion or ask your own question.

Linked
5 Why is unsigned 4 not considered greater than signed -2?
5 Integer promotion example in C
1 Overflow of integer calculation in C and C++
3 How come I can compare a char with an int?
2 unexpected value when assign unsigned int(by negtive it) to signed long long
3 C(++) short subtracting short = int?
0 The type returned by 'operator|' does not always match the type you passed it
1 Why multiplying int and char doesn't produce overflow in c?
0 Shouldn't the difference of two uint16_t types also be a uint16_t?
2 Subtracting long int and unsigned int
See more linked questions

Related
2 Managing Implicit Type Conversion in C++
1 Implicit Type Conversion
2 c++ implicit conversion rules
0 Implicit types for numbers in C
4 explicit/implicit type conversion c++
14 Inconsistent behaviour of implicit conversion between unsigned and bigger signed types
3 Confused by effective type rules
1 Why should be there the involvement of type promotion in this code?
1 Unexpected Result for Implicit Type Conversion
1 Understanding promotions (type conversions)

Hot Network Questions
What is the German equivalent of "ing"
What does it mean for the laws of physics to "break down" at a singularity?
Origin of the phrase "crazy as a coon"—is it racist?
How did Professor Sprout bandage the Whomping Willow?
Why are segmented mirrors lighter than monolithic mirrors?
What is the right age to teach your kids about weather-appropriate clothing?
Somebody sent me a check to deposit into my account, but I now think that it's a scam. will I get in trouble?
Is there a way to get file description field from the file metadata using the command line?
How can I renew a Belarusian passport abroad?
What is an ideal layout for a normally-open solder jumper?
Does this expression always vanish?
Remove indent of paragraph after question in exam document class
How can I find out what % of the health insurance premium my employer pays?
How long would it take for a Japanese-style castle to burn to the ground?
Am I Doing It Wrong? Why Can't I Use Multiple 5-Line Staves to Represent Guitar's Open Strings?
Career change: Entering Academia realistic at 36?
What can be found on the "terminal" on the Jurassic World promotional website?
Example about endomorphism ring of an indecomposable object, which have non-trivial idempotent element in an additive category.
Orion stars distance from Earth
The world's smallest square maze?
What happens if you ignore a howler?
Apple IIc making grinding sounds when accessing disks beyond normal clicks
Implement Subleq
Why is the ECMAScript specification formatted the way it is?
Question feed