

Forum:[Hack Forums](#)

Thread: [C - when to use \(long long\) before the operation?](#)

FaberCastell - September 6, 2017, 7:23 am

So my question is, how do i know when do i have to use (long long), or (long) or any other data type before the operation... I don't understand what's the point and when to use it.

In the example below, without (long long) the result ended up somehow being negative and incorrect.

```
Code unsigned long long myNumber;
```

```
(loop)
myNumber+= (long long) pow(2, i);
(endofloop)
```

Trip2nite - September 6, 2017, 8:55 am

It is a question of the size of datatype.

Unsigned long long can hold value from 0 - 18446744073709551615.

Unsigned long can hold value from: 0 - 4294967295

https://en.wikipedia.org/wiki/C_data_types

FaberCastell - September 6, 2017, 9:44 am

(Sep 6, 2017 08:55 AM)Trip2nite Wrote: [\[CITE\]](#) It is a question of the size of datatype.

Unsigned long long can hold value from 0 - 18446744073709551615.

Unsigned long can hold value from: 0 - 4294967295

https://en.wikipedia.org/wiki/C_data_types

yeah i know that, but the question is why do i need to put (long long) before pow(2, i)

myNumber is already unsigned long long, why is needed then

```
myNumber+= (long long) pow(2,i);
```

Trip2nite - September 6, 2017, 10:17 am

(Sep 6, 2017 09:44 AM)FaberCastell Wrote: [\[CITE\]](#) yeah i know that, but the question is why do i need to put (long long) before pow(2, i)

myNumber is already unsigned long long, why is needed then

```
myNumber+= (long long) pow(2,i);
```

Well, IIRC pow (in math.h) returns a double.

You are essentially just casting it from double to long long.

Infinity. - September 6, 2017, 11:38 am

(Sep 6, 2017 10:17 AM)Trip2nite Wrote: [\[CITE\]](#) Well, IIRC pow (in math.h) returns a double.

You are essentially just casting it from double to long long.

His variable is unsigned long long and he used long long for casting. There was no C89 support for long long but MSVC supports it.

0xDEAD10CC - September 6, 2017, 12:10 pm

The answer can probably be easily found if you look into IEEE754, two's complement (signed vs unsigned).. But even then I'm still unsure on how or WHY you think the result would be negative when your storage type is unsigned. Perhaps you were calling printf() with a "signed" specifier like %d incorrectly, when it should have been %u with the proper length (%llu)?

FaberCastell - September 6, 2017, 12:49 pm

(Sep 6, 2017 12:10 PM)0xDEAD10CC Wrote: [\[CITE\]](#) The answer can probably be easily found if you look into IEEE754, two's complement (signed vs unsigned).. But even then I'm still unsure on how or WHY you think the result would be negative when your storage type is unsigned. Perhaps you were calling printf() with a "signed" specifier like %d incorrectly, when it should have been %u with the proper length (%llu)?

Could be that. My point though was when do i have to use (long long) and someone mentioned that pow returns a double so that's what i wanted. Do all operations from math.h return a double or how do i know when do i have to cast another data type?

0xDEAD10CC - September 6, 2017, 1:11 pm

I don't understand your question, have you read the responses? Different datatypes have different limits and precision. You determine what you need based on what you're doing and the types returned by functions, simple.

FaberCastell - September 6, 2017, 1:17 pm

Yes but i set datatype unsigned long long myNumber

but why do i have to do then myNumber+=(long long) pow(2,i)

I have already assigned "unsigned long long" to myNumber - then why do i have to do (long long) before pow.

I don't understand when i have to do this (long long) again when my integer is already unsigned long long - how can you determine if you have to or not?

0xDEAD10CC - September 6, 2017, 1:20 pm

Why do you think you need to? I already told you - you're not going to get a negative result from an unsigned type. When you cast things you have the potential for narrowing and widening, but the more problematic conversion is a narrowing one. Primitive integers do not have decimals like floating point types either.

Also, if you understand floating point error you would also know why people use extreme caution when using floating point types for precision. When dealing with larger integers a long long is going to support much larger whole numbers too...

Code int main(void)

```
{
    float x = 0.;
    for (float i = .10; i < 100.; i += .10)
    {
        x += i;
    }
    printf("%f\n", x);
    return 0;
}
```

The result might not be what you expect.

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Ronnie Coleman - September 8, 2017, 9:53 am

(Sep 6, 2017 07:23 AM)FaberCastell Wrote: [\[CITE\]](#) So my question is, how do i know when do i have to use (long long), or (long) or any other data type before the operation... I don't understand what's the point and when to use it.

In the example below, without (long long) the result ended up somehow being negative and incorrect.

```
Code unsigned long long myNumber;
```

```
(loop)
myNumber+= (long long) pow(2, i);
(endofloop)
```

1. Use bitshifts for computing powers of two. You can do left bitshifts using << operator. For example, 2^{18} (i.e. 2 to the power of 18) using $2 \ll 18$. The bitshift operation will use lesser clock cycle, avoid inaccuracy, and will not use the FPU.
2. Initialize myNumber to 0, the default value is compiler-dependent, you don't want that.
3. You should identify the largest power you want to handle. For powers ≤ 32 , use unsigned int. For powers ≤ 64 , use unsigned long long int. If you want your code to be really portable, you can use $\text{sizeof}(\text{TYPE}) * 8$ to compute the number of bits in a integer TYPE.

ps: sizeof() macro returns the number of bytes in a data type.

I get your question. Try setting myNumber to 0. I think it will be corrected after doing that. LMK otherwise.

(Sep 6, 2017 01:20 PM)0xDEAD10CC Wrote: [\[CITE\]](#) Why do you think you need to? I already told you - you're not going to get a negative result from an unsigned type. When you cast things you have the potential for narrowing and widening, but the more problematic conversion is a narrowing one. Primitive integers do not have decimals like floating point types either.

Also, if you understand floating point error you would also know why people use extreme caution when using floating point types for precision. When dealing with larger integers a long long is going to support much larger whole numbers too...

```
Code int main(void)
{
    float x = 0.;
    for (float i = .10; i < 100.; i += .10)
    {
        x += i;
    }
    printf("%f\n", x);
    return 0;
}
```

The result might not be what you expect.

https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Last statement is not necessarily true. The point of floating points is representing large numbers with a sacrifice in accuracy. You can represent extremely large integers with double,

0xDEAD10CC - September 9, 2017, 12:57 pm

(Sep 8, 2017 09:53 AM)_Russ Wrote: [\[CITE\]](#) 1. Use bitshifts for computing powers of two. You can

do left bitshifts using << operator. For example, 2^{18} (i.e. 2 to the power of 18) using $2 \ll 18$. The bitshift operation will use lesser clock cycle, avoid inaccuracy, and will not use the FPU.

2. Initialize myNumber to 0, the default value is compiler-dependent, you don't want that.

3. You should identify the largest power you want to handle. For powers ≤ 32 , use unsigned int. For powers ≤ 64 , use unsigned long long int. If you want your code to be really portable, you can use `sizeof(TYPE) * 8` to compute the number of bits in a integer TYPE.

ps: `sizeof()` macro returns the number of bytes in a data type.

I get your question. Try setting myNumber to 0. I think it will be corrected after doing that. LMK otherwise.

Last statement is not necessarily true. The point of floating points is representing large numbers with a sacrifice in accuracy. You can represent extremely large integers with double,

1. `sizeof()` is NOT a macro once again.

2. You shouldn't be hardcoding 8, you should use `CHAR_BIT` instead from `<limit.h>`

3. Which "last" statement? Is this not just what I finished saying:

Quote: floating points is representing large numbers with a sacrifice in accuracy

You can't represent extremely large integers with any floating point type because it is not an integer datatype. If you understand IEEE 754 they are represented from integers in the background.

Ronnie Coleman - September 10, 2017, 7:28 am

(Sep 9, 2017 12:57 PM)0xDEAD10CC Wrote: [CITE](#) 1. `sizeof()` is NOT a macro once again.

2. You shouldn't be hardcoding 8, you should use `CHAR_BIT` instead from `<limit.h>`

3. Which "last" statement? Is this not just what I finished saying:

You can't represent extremely large integers with any floating point type because it is not an integer datatype. If you understand IEEE 754 they are represented from integers in the background.

Point taken. I understand IEEE 754 perfectly. I misunderstood your point.

I just meant to say that certain large numbers cannot be represented by unsigned 64-bit integer but can be represented by double. Probably not worth discussing.