1-floating-point-basics

June 3, 2024

1 Floating point basics

Floating-point numbers can represent a very large range of numbers, from the smallest to the largest, similarly to scientific notation. They are the prefered types for scientific computing. Yet, one must be aware of the many **rounding errors** which are implied.

When such a large range is not useful, one can use fixed-point numbers to obtain faster calculations. When looking for maximum precision, while accepting a longer execution time, one can rely on quotients of integers.

1.1 General representation of a floating-point number

In a given base b, which will be the same for all the numbers, the representation of a number includes: * one bit for the sign (equal to -1 or 1), * the mantissa, * the exponent (relative integer).

The encoded number is equivalent to *sign* $\times$ *mantissa* $\times$ *b^exponent*

When the exponent vary, the comma somehow "float", hence the name. The mantissa is a sequence of digits in base b, generally of fixed size, in which we choose to place a comma at a given position: just before or just after the first digit, or even just after the last digit (in this case the mantissa is a natural number).

Floating-point numbers are a kind of **computer equivalent of scientific notation**, The later rather using a base of 10 and placing the comma after the first digit of the mantissa. Over time several computing variants have emerged, inducing problems of portability. This motivated the creation of the IEEE 754 standard.

1.2 IEEE 754 standard

In 1984 the IEEE (Institute of Electrical and Electronics Engineers) defined its standard * IEEE 754 *, which was improved in 1987, 2008, and recently in July 2019.

The standard proposes different formats in base 2 and in base 10. For base 2, it imposes that the comma is placed after the first bit of the mantissa. Since, in base `2`, the first bit is always`1`, we do not store it: it is "implicit". For the rest, the standard distributes the bits as follows:

Name	Common name	Size	Sign	Exponent	Mantissa	Significant decimal digits
binary16	Half precision	16 bits	1 bit	5 bits	10 bits	?
binary32	Single precision	32 bits	1 bit	8 bits	23 bits	about 7
binary64	Double precision	64 bits	1 bit	11 bits	52 bits	about 16

Name	Common name	Size	Sign	Exponent	Mantissa	Significant decimal digits
binary128	Quadruple precision	128 bits	1 bit	15 bits	112 bits	?

1.3 Predefined floating-point types in C++

The standard does not enforce size of floating-point types. This can cause portability issues from one machine to another. The only guarantee that the standard provides us is

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

However, nowadays, we can assume that all the architectures use the format **IEEE 754 binary32 for the type `float`**, and the format **IEEE 754 binary64 for the type`double`** and it should come with an efficient hardware implementation for operations on these numbers.

The situation varies more widely for `long double`. If the size of the type on your architecture is 16 bytes (128 bits), it is common that only the first 80 bits are used for the computation, at least for an Intel processor.

It is also possible that some operations are not directly supported by the hardware, but rather "software-emulated", with reduced performance.

To check for possible portability issues, you can add `static_assert` into your code in order to verify that the size of the types is what you expect.

If you restrain yourself to the g++ compiler, and accept its extensions to the standard, g++ offers you either `__float80` or`__float128` types, depending on the architecture you are compiling on.

1.4 Exotic precisions

If the level of precision of `long double` is not enough for you, there are few ways to explore (below). Of course, as there is no direct hardware support, the execution times may be a lot longer. * the GNU library **MPFR** allows one to choose an arbitrary precision. * of course, as for many other problems, there is the Boost library :) * some algorithms of **compensated arithmetic** make it possible to limit the losses linked to rounding during a calculation.

In addition, especially since the great comeback of machine learning, there is a growing need for numbers with lower precision: * The **half-precision**, which is stored in 2 bytes, will also rely on an external library and will only be effective on certain specific hardware such as GPGPUs; * **BF16** is a variant of the previous one which uses more bits for the exponent and less for the mantissa and which therefore favors "order of magnitude" to the detriment of precision.

It should also be mentioned that the **80-bit extended precision** available on Intel processors was used in the intermediate calculations even though the start and end data were `double`. On the contrary, it was not used for vectorized instructions, which led to different results depending on whether a code was vectorized or not. As far as we know, compilers now avoid this use of 80 bits, even for scalar computations, unless explicitly asked for.

1.5 New in C++23 : fixed width floating-point types

implementation dependant

If the implementation supports any of the following ISO 60559 types as an extended floating-point type, then: - the corresponding macro is defined as 1 to indicate support, - the corresponding floating-point literal suffix is available, and - the corresponding type alias name is provided:

Type name Defined in header	Literal suffix	Predefined macro	bits of storage	bits of precision	bits of exponent	max exponent
std::float16_t	f16 or F16	**STDCPP_FLOAT16_T**16	11	5	15	
std::float32_t	f32 or F32	**STDCPP_FLOAT32_T**32	24	8	127	
std::float64_t	f64 or F64	**STDCPP_FLOAT64_T**64	53	11	1023	
std::float128_t	f128 or F128	**STDCPP_FLOAT128_T**28	113	15	16383	
std::bfloat16_t	bf16 or BF16	**STDCPP_BFLOAT16_T**16	8	8	127	

2 Questions ?

3 Exercise

Using the numeric traits defined via std::numeric_limits in the standard library, complete the following table:

Type	Size (in bytes)	Minimal value	Maximum value	Epsilon
float				
double				
long double				

To make sure you don't lose any information while printing out, make sure you display 18 significant digits. * What do we mean here by "minimal value" ? What is the difference with the "lowest value" ? * What do you think of the results for `long double`: it is rather 80 or 128 bits ?

If your compiler support it, try the same with the new fixed width floating-point types:

Type	Size (in bytes)	Minimal value	Maximum value	Epsilon
std::float16_t				
std::bfloat16_t				
std::float32_t				
std::float64_t				
std::float128_t				

3.1 Sources

- LearnCpp
- Interflop

- Wikipedia
- Cpp Reference
- Fixed width floating-point types
- QuadMath
- What Every Programmer Should Know About Floating-Point Arithmetic
- IEEE-754 Floating-Point Conversion