5.3. Numeric Limits

Numeric types in general have platform-dependent limits. The C++ standard library provides these limits in the template `numeric_limits`. These numeric limits replace and supplement the ordinary preprocessor C constants, which are still available for integer types in `<climits>` and `<limits.h>` and for floating-point types in `<cfloat>` and `<float.h>`. The new concept of numeric limits has two advantages: First, it offers more type safety. Second, it enables a programmer to write templates that evaluate these limits.

The numeric limits are discussed in the rest of this section. Note, however, that it is always better to write platform-independent code by using the minimum guaranteed precision of the types. These minimum values are provided in Table 5.8.[19]

[19] Note that "bytes" means an octet with 8 bits. Strictly speaking, it is possible that even a `long int` has one byte with at least 32 bits.

Table 5.8. Minimum Size of Built-In Types

Type	Minimum Size
`char`	1 byte (8 bits)
`short int`	2 bytes
`int`	2 bytes
`long int`	4 bytes
`long long int`	8 bytes
`float`	4 bytes
`double`	8 bytes
`long double`	8 bytes

Class `numeric_limits<>`

You usually use templates to implement something once for any type. However, you can also use templates to provide a common interface that is implemented for each type, where it is useful. You can do this by providing specializations of a general template. A typical example of this technique is `numeric_limits`, which works as follows:

- A general template provides the default numeric values for any type:

Click here to view code image

```
namespace std {
    // general numeric limits as default for any type
    template <typename T>
    class numeric_limits {
      public:
        // by default no specialization for any type T exists
        static constexpr bool is_specialized = false;
        ...  // other members are meaningless for the general template
    };
}
```

This general template of the numeric limits says that no numeric limits are available for type `T`. This is done by setting the member `is_specialized` to `false`.

- Specializations of the template define the numeric limits for each numeric type as follows:

```
namespace std {
    // numeric limits for int
    // - implementation defined
    template<> class numeric_limits<int> {
      public:
        // yes, a specialization for numeric limits of int does exist
        static constexpr bool is_specialized = true;

        static constexpr int min() noexcept {
            return -2147483648;
        }
        static constexpr int max() noexcept {
```

```
                return 2147483647;
            }
            static constexpr int digits = 31;
            …
        };
    }
```

Here, `is_specialized` is set to `true`, and all other members have the values of the numeric limits for the particular type.

The general `numeric_limits` template and its standard specializations are provided in the header file `<limits>`. The specializations are provided for any fundamental type that can represent numeric values: `bool`, `char`, `signed char`, `unsigned char`, `char16_t`, `char32_t`, `wchar_t`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double`, and `long double`.[20] They can be supplemented easily for user-defined numeric types.

[20] The specializations for `char16_t`, `char32_t`, `long long`, and `unsigned long long` are provided since C++11.

Tables 5.9 and 5.10 list all members of the class `numeric_limits<>` and their meanings. Applicable corresponding C constants for these members, defined in `<climits>`, `<limits.h>`, `<cfloat>`, and `<float.h>`, are also given.

Table 5.9. Members of Class *numeric_limits<>*, Part 1

Member	Meaning	C Constants
`is_specialized`	Type has specialization for numeric limits	
`is_signed`	Type is signed	
`is_integer`	Type is integer	
`is_exact`	Calculations produce no rounding errors (`true` for all integer types)	
`is_bounded`	The set of values representable is finite (`true` for all built-in types)	
`is_modulo`	Adding two positive numbers may wrap to a lesser result	
`is_iec559`	Conforms to standards IEC 559 and IEEE 754	
`min()`	Minimum finite value (minimum positive normalized value for floating-point types with denormalization; meaningful if `is_bounded||!is_signed`)	`INT_MIN`,`FLT_MIN`, `CHAR_MIN`,...
`max()`	Maximum finite value (meaningful if `is_bounded`)	`INT_MAX`,`FLT_MAX`,...
`lowest()`	Maximum negative finite value (meaningful if `is_bounded`; since C++11)	
`digits`	Character/integer: number of bits, excluding sign (binary digits)	`CHAR_BIT`

	Floating point: number of `radix` digits in the mantissa	`FLT_MANT_DIG,...`
`digits10`	Number of decimal digits (meaningful if `is_bounded`)	`FLT_DIG,...`
`max_digits10`	Number of required decimal digits to ensure that values that differ are always differentiated (meaningful for all floating-point types; since C++11)	
`radix`	Integer: base of the representation (almost always 2)	
	Floating point: base of the exponent representation	`FLT_RADIX`
`min_exponent`	Minimum negative integer exponent for base `radix`	`FLT_MIN_EXP,...`
`max_exponent`	Maximum positive integer exponent for base `radix`	`FLT_MAX_EXP,...`
`min_exponent10`	Minimum negative integer exponent for base 10	`FLT_MIN_10_EXP,...`
`max_exponent10`	Maximum positive integer exponent for base 10	`FLT_MAX_10_EXP,...`
`epsilon()`	Difference of 1 and least value greater than 1	`FLT_EPSILON,...`
`round_style`	Rounding style (see page 119)	
`round_error()`	Measure of the maximum rounding error (according to standard ISO/IEC 10967-1)	
`has_infinity`	Type has representation for positive infinity	
`infinity()`	Representation of positive infinity, if available	
`has_quiet_NaN`	Type has representation for nonsignaling "Not a Number"	
`quiet_NaN()`	Representation of quiet "Not a Number," if available	

Table 5.10. Members of Class `numeric_limits<>`, Part 2

Member	Meaning	C Constants
`has_signaling_NaN`	Type has representation for signaling "Not a Number"	
`signaling_NaN()`	Representation of signaling "Not a Number," if available	
`has_denorm`	Whether type allows denormalized values (variable numbers of exponent bits; see page 119)	
`has_denorm_loss`	Loss of accuracy is detected as a denormalization loss rather than as an inexact result	
`denorm_min()`	Minimum positive denormalized value	
`traps`	Trapping is implemented	
`tinyness_before`	Tinyness is detected before rounding	

The following is a possible full specialization of the numeric limits for type `float`, which is platform dependent and shows the exact signatures of the members:

Click here to view code image

```
namespace std {
   template<> class numeric_limits<float> {
     public:
       // yes, a specialization for numeric limits of float does exist
       static constexpr bool is_specialized = true;

       inline constexpr float min() noexcept {
           return 1.17549435E-38F;
       }
       inline constexpr float max() noexcept {
           return 3.40282347E+38F;
       }
       inline constexpr float lowest() noexcept {
           return -3.40282347E+38F;
       }

       static constexpr int digits = 24;
       static constexpr int digits10 = 6;
       static constexpr int max_digits10 = 9;

       static constexpr bool is_signed = true;
```

```
        static constexpr bool is_integer = false;
        static constexpr bool is_exact = false;
        static constexpr bool is_bounded = true;
        static constexpr bool is_modulo = false;
        static constexpr bool is_iec559 = true;
        static constexpr int radix = 2;

        inline constexpr float epsilon() noexcept {
            return 1.19209290E-07F;
        }

        static constexpr float_round_style round_style
            = round_to_nearest;
        inline constexpr float round_error() noexcept {
            return 0.5F;
        }

        static constexpr int min_exponent = -125;
        static constexpr int max_exponent = +128;
        static constexpr int min_exponent10 = -37;
        static constexpr int max_exponent10 = +38;

        static constexpr bool has_infinity = true;
        inline constexpr float infinity() noexcept { return  ...; }
        static constexpr bool has_quiet_NaN = true;
        inline constexpr float quiet_NaN() noexcept { return  ...; }
        static constexpr bool has_signaling_NaN = true;
        inline constexpr float signaling_NaN() noexcept { return  ...; }
        static constexpr float_denorm_style has_denorm = denorm_absent;
        static constexpr bool has_denorm_loss = false;
        inline constexpr float denorm_min() noexcept { return min(); }

        static constexpr bool traps = true;
        static constexpr bool tinyness_before = true;
    };
}
```

Note that since C++11, all members are declared as `constexpr` (see Section 3.1.8, page 26). For example, you can use `max()` at places where compile-time expressions are required:

```
static const int ERROR_VALUE = std::numeric_limits<int>::max();
float a[std::numeric_limits<short>::max()];
```

Before C++11, all nonfunction members were constant and static, so their values could be determined at compile time. However, function members were `static` only, so the preceding expressions were not possible. Also note that before C++11, `lowest()` and `max_digits10` were not provided and that empty exception specifications instead of `noexcept` (see Section 3.1.7, page 24) were used.

The values of `round_style` are shown in Table 5.11. The values of `has_denorm` are shown in Table 5.12. Unfortunately, the member `has_denorm` is not called `denorm_style`. This happened because during the standardization process, there was a late change from a Boolean to an enumerative value. However, you can use the `has_denorm` member as a Boolean value because the standard guarantees that `denorm_absent` is `0`, which is equivalent to `false`, whereas `denorm_present` is `1` and `denorm_indeterminate` is `-1`, both of which are equivalent to `true`. Thus, you can consider `has_denorm` a Boolean indication of whether the type may allow denormalized values.

Table 5.11. Round Style of `numeric_limits<>`

Round Style	Meaning
`round_toward_zero`	Rounds toward zero
`round_to_nearest`	Rounds to the nearest representable value
`round_toward_infinity`	Rounds toward positive infinity
`round_toward_neg_infinity`	Rounds toward negative infinity
`round_indeterminate`	Indeterminable

Table 5.12. Denormalization Style of `numeric_limits<>`

Denorm Style	Meaning
`denorm_absent`	The type does not allow denormalized values
`denorm_present`	The type allows denormalized values to the nearest representable value
`denorm_indeterminate`	Indeterminable

Example of Using `numeric_limits<>`

The following example shows possible uses of some numeric limits, such as the maximum values for certain types and determining whether `char` is signed:

Click here to view code image

```
// util/limits1.cpp

#include <iostream>
#include <limits>
#include <string>
using namespace std;

int main()
{
    // use textual representation for bool
    cout << boolalpha;

    // print maximum of integral types
    cout << "max(short): " << numeric_limits<short>::max() << endl;
    cout << "max(int):   " << numeric_limits<int>::max() << endl;
    cout << "max(long):  " << numeric_limits<long>::max() << endl;
    cout << endl;

    // print maximum of floating-point types
    cout << "max(float):       "
         << numeric_limits<float>::max() << endl;
    cout << "max(double):      "
         << numeric_limits<double>::max() << endl;
    cout << "max(long double): "
         << numeric_limits<long double>::max() << endl;
    cout << endl;

    // print whether char is signed
    cout << "is_signed(char): "
         << numeric_limits<char>::is_signed << endl;
    cout << endl;

    // print whether numeric limits for type string exist
    cout << "is_specialized(string): "
         << numeric_limits<string>::is_specialized << endl;
}
```

The output of this program is platform dependent. Here is a possible output of the program:

```
max(short): 32767
max(int):   2147483647
max(long):  2147483647

max(float):       3.40282e+38
max(double):      1.79769e+308
max(long double): 1.79769e+308

is_signed(char): false

is_specialized(string): false
```

The last line shows that no numeric limits are defined for the type `string`. This makes sense because strings are not numeric values. However, this example shows that you can query for any arbitrary type whether or not it has numeric limits defined.