

$$V = (-1)^{b_f+i} \left(\sum_{n=0}^i 2^n b_{n+f} + \sum_{m=1}^f 2^{-m} b_{f-m} \right).$$

Here is an example: assume that we have a 16 bit fixed point number, in 7.8 format with an additional sign bit, where the bits are set to 0000 0110 1010 0000 (that is 0x06A0 in hexadecimal notation). The value of this fixed point number is +6.625.

If you have such a bit pattern, regardless of the fixed point format, you can treat it as an integer and perform integer mathematics on it. Only one little complication arises when you do multiplication or division: you need to make sure that the decimal point is at the right position. This is roughly comparable to the way you learned to multiply by hand in school: at the end of the multiplication you had to count positions after the decimal point and then set the decimal point at the right position.

The implication for the machine is that fast integer commands can be used to carry out the calculations. The downside is that range and precision of fixed point numbers are usually smaller than those of floating point numbers.

Usage of the fixed_point Class

The first thing you should do is to include the header file `fixed_point.h`:

```
#include <fixed_point.h>
```

The `fixed_point<B, I, F>` class is defined inside the namespace `fpml`. In order to use it, you can either prefix with `fpml::` everywhere it is necessary, or you can use a using statement:

```
using namespace fpml;
```

There are basically two different use-cases for the `fixed_point<B, I, F>` class. You can either use it in newly written code, where you know that you want fixed-point mathematics, and where you control the behavior, i.e. determine the number of integer and fractional bits. An extremely simple code sample of this first use case could look like this:

```
#include <fixed_point.h>
using namespace fpml;

main()
{
    fixed_point<int, 16> a = 256;
    fixed_point<int, 16> b = sqrt(a);
}
```

Of course, you can use all the other operators and functions as well. The `fixed_point<B, I, F>` class has implementations for all the important operators and functions you would take for granted when working with floating point numbers.

The second use case is the scenario of porting, when the code already exists and has been written with either the float or double types. When you have carefully checked the code for range and

precision issues and have decided that the code will still work when the floating point calculations are replaced with fixed point calculations, you can use `#define` to port the code with minor changes made to the original code:

```
#include <fixed_point.h>
#define double fpml::fixed_point<int, 16>

... original code here, unchanged ...

#undef double
```

You should be very careful though, because both range and precision of the fixed-point numbers will differ from those of the original floating-point numbers, and this may introduce bugs into the original code. Especially when functions like `sqrt`, `log` or `sin` are used, the error propagation of the fixed-point numbers is no longer linear and surprises may be encountered.

Implementation of the fixed_point Class

One of the goals is that the code should be as generic as possible. One way to achieve this is to use templates. I decided to use three template parameters:

```
template<typename B, unsigned char I, unsigned char F
    = std::numeric_limits<B>::digits - 1>
class fixed_point
{
    BOOST_CONCEPT_ASSERT((boost::Integer<B>));
    BOOST_STATIC_ASSERT(I + F == std::numeric_limits<B>::digits);
    ...
private:
    B value_;
}
```

`B` is the base type. This must be an integer type. This is the type used for most of the calculations. Examples are `unsigned char`, `signed short`, or `int`. This type can be chosen according to the size, precision and performance requirements. If an unsigned type is chosen, the behavior of the `fixed_point` class is unsigned, otherwise it is signed. Signed behavior more closely matches the behavior of the built-in floating point types.

The statement `BOOST_CONCEPT_ASSERT((boost::Integer))` in the class body assures that only integer types can be used as the base type. Please note that the base type is not used in the sense of base class here. In fact the `fixed_point` class is not derived from any base class, but stands on its own feet.

`I` is the number of bits of the integer part, not counting the sign bit. This determines the range of the numbers that can be represented.

`F` is the number of bits of the fractional part. This determines the precision of the numbers that can be represented. There is no need to specify `F` when the template is instantiated, since it can always be deferred automatically from the base type `B` and the number of integer bits `I`. However, if it is

specified then it needs to be correct. The statement `BOOST_STATIC_ASSERT(I + F == std::numeric_limits::digits)` in the class body assures that this condition is enforced.

The number of bits of the base type must satisfy the following formula:

$$\#(B) = S + I + F$$

where

- $\#(B)$ is the number of bits of the base type,
- S is 1 for signed types and 0 for unsigned types,
- I is the number of integer bits,
- F is the number of fractional bits.

The table shows the usable types and the requirements for I and F :

B	$\#(B)$	S	I	F
signed char	8	1	7...1	0...6
unsigned char	8	0	8...1	0...7
short	16	1	15...1	0...14
unsigned short	16	0	16...1	0...15
int	32	1	31...1	0...30
unsigned int	32	0	32...1	0...31

Objects of type `fixed_point<B, I, F>` are of the same size as the underlying type `B`. The class has been carefully designed not to impose additional size requirements.

By using all available bits for the integer part, you achieve integer behavior. However, you can call the defined functions such as `sqrt` or `log` etc. for integers then. I have more to say about these functions later.

Types larger than 32 bit are not yet supported. One reason for this is that some functions need internal results twice as big. If 64 bit types would have been allowed, these internal results would be 128 bit wide. The second reason is that if you could afford 64 bits for the fixed-point type, you could as well use type `double` in many cases.

Construction and Conversion

If an object of type `fixed_point<B, I, F>` should be usable, it first needs to be constructed. The class provides a set of constructors. First, there is the parameter-less default constructor:

```
fixed_point()
{ }
```

Just as with built-in types no initialization is done. The value is undetermined after executing this constructor.

Second, there is a constructor which allows construction from integer values. This is implemented with a template:

```
template<typename T> fixed_point(T value) : value_((B)value << F)
{
    BOOST_CONCEPT_ASSERT((boost::Integer<T>));
}
```

This constructor takes an integer value of type `T` and converts it to this `fixed_point<B, I, F>` type. The conversion from the integer format to the fixed point format is done by shifting `F` positions to the left, so that the position of the binary point is correct.

Third, there is a constructor which allows construction from a boolean value:

```
fixed_point(bool value) : value_((B)(value * power2<F>::value))
{ }
```

A `fixed_point<B, I, F>` object constructed from `false` has a value of `0.0`, a `fixed_point<B, I, F>` object constructed from `true` has a value of `1.0`.

Fourth, there are some constructors which allow construction from floating point values:

```
fixed_point(float value) : value_((B)(value * power2<F>::value))
{ }

fixed_point(double value) : value_((B)(value * power2<F>::value))
{ }

fixed_point(long double value) : value_((B)(value * power2<F>::value))
{ }
```

These constructors take a floating point value of type `float`, `double` or `long double` and convert it to the `fixed_point<B, I, F>` type. The conversion is done by multiplying with an appropriate power of 2 and casting to the result to base type `B`. The appropriate power of 2 is determined by the number of bits `F` of the fractional part. This corresponds to the shift operation performed by the integer constructors.

All the constructors so far with one parameter also serve as implicit conversion operators, and convert from the type of the constructor parameter to the `fixed_point<B, I, F>` type. Therefore, you can initialize `fixed_point<B, I, F>` variables with numeric values, as follows:

```
fixed_point<int, 16> a(0);
fixed_point<int, 16> b = -1.5;
```

Finally, a copy constructor is implemented:

```
fixed_point(fixed_point<B, I, F> const& rhs) : value_(rhs.value_) { }
```

The copy constructor simply copies the content of the `fixed_point<B, I, F>::value_` member.

Strictly seen this constructor is not necessary, since the compiler would automatically synthesize a similar copy constructor. However, I like to make things explicit.

As mentioned before, the constructors with one parameter doubly serve as conversions from numeric values to the `fixed_point<B, I, F>` type. The other direction of the conversion is also needed and is implemented with conversion operators, for the same numeric types:

```
template<typename T> operator T() const
{
    BOOST_CONCEPT_ASSERT((boost::Integer<T>));
    return value_ >> F;
}

operator float() const
{
    return (float)value_ / power2<F>::value;
}

operator double() const
{
    return (double)value_ / power2<F>::value;
}

operator long double() const
{
    return (long double)value_ / power2<F>::value;
}
```

The conversions from type `fixed_point<B, I, F>` are symmetric to the constructors. As such the integer conversions shift to the right, and the floating point conversions divide by an appropriate power of 2.

There are a few other remarks to be made here.

Sadly, C++ does not specify the exact behaviour of the shift operators, but leaves it implementation defined. Any implementation is free to do an arithmetic shift (correctly handling the sign bit for negative numbers) or a logic shift (not handling the leftmost bit specially). This has the potential that the code may not work as intended. However, the implementations I tried (Visual Studio 2005, Visual Studio 2008) do the right thing: they do an arithmetic shift for signed numbers and they do a logic shift for unsigned numbers.

The floating point conversions need to calculate 2 to the power of `F`. I could have used the runtime function `pow`, but I did not want to defer a calculation to runtime that could equally well be done at compile time. Unfortunately it is not so straightforward. I used a template metaprogramming technique I've seen somewhere (but I don't remember where) in order to calculate 2^F at compile time:

```
template<int F> struct power2 {
    enum { value = 2 * power2<F-1>::value };
};
template <> struct power2<0> {
    enum { value = 1 };
};
```

The `power2` template works by template recursion. For example, if `F == 2`, the following steps are carried out:

1. `power2<2>` is instantiated, `power2<2>::value` is set to `2 * power2<1>::value`. Since the `power2<1>::value` is yet unknown, the compiler needs to instantiate `power2<1>`.
2. `power2<1>` is instantiated, `power2<1>::value` is set to `2 * power2<0>::value`. Since the `power2<0>::value` is yet unknown, the compiler needs to instantiate `power2<0>`.
3. `power2<0>` is instantiated, `power2<0>::value` is 1. Now the recursion can wind the whole way back, effectively calculating `2 * 2 * 1`, which is 2^2 equalling 4.

Operators

Of course, to be able to do something useful with objects of type `fixed_point<B, I, F>`, some operators are needed.

Assignment and Conversion

There is a simple assignment operator, which is implemented in terms of the copy constructor and the swap operation.

```
fixed_point<B, I, F> & operator =(fixed_point<B, I, F> const& rhs)
{
    fixed_point<B, I, F> temp(rhs);
    swap(temp);
    return *this;
}
```

The swap operation itself delegates to the swap of the C++ standard library.

```
void swap(fixed_point<B, I, F> & rhs)
{ std::swap(value_, rhs.value_); }
```

There is also a version of the assignment which can convert between different fixed-point formats, and consequently also a converting copy constructor is needed. Conversion between different fixed-point formats can be done by shifting the representation left or right as needed.

```
template<unsigned char I2, unsigned char F2>
fixed_point<B, I, F> & operator =(fixed_point<B, I2, F2> const& rhs)
{
    fixed_point<B, I, F> temp(rhs);
    swap(temp);
    return *this;
}

template<unsigned char I2, unsigned char F2>
fixed_point(fixed_point<B, I2, F2> const& rhs)
    : value_(rhs.value_)
{
    if (I-I2 > 0)
        value_ >>= I-I2;
    if (I2-I > 0)
        value_ <<= I2-I;
}
```

Comparison

For comparison, `operator <` and `operator ==` are implemented.

```
bool operator <(fixed_point<B, I, F> const& rhs) const
{
    return value_ < rhs.value_;
}

bool operator ==(fixed_point<B, I, F> const& rhs) const
{
    return value_ == rhs.value_;
}
```

The `boost::ordered_field_operators` class automatically implements `operator >`, `operator >=` and `operator <=` in terms of `operator <`, as well as `operator !=` in terms of `operator ==`.

In pseudo-code notation this automatic provision of operators looks like this:

```
boost::ordered_field_operators
operator <(a, b)
operator >(a, b): return operator <(b, a);
operator >=(a, b): return ! operator <(a, b);
operator <=(a, b): return ! operator <(b, a);
operator ==(a, b)
operator !=(a, b): return ! operator ==(a, b);
```

Conversion to bool

Floating point types `float` and `double` support conversion to `bool`. A floating-point value converts to `true` when it is `!= 0` and it converts to `false` otherwise. Thus, I have implemented a conversion to `bool` and `operator !`, which returns just the inverse.

```
operator bool() const
{
    return (bool)value_;
}

bool operator !() const
{
    return value_ == 0;
}
```

Unary operator -

For signed fixed-point types you can apply the unary minus operator to get the additive inverse. For unsigned fixed-point types, this operation is undefined. Also, shared with the integer base type `B`, the minimum value representable by the type cannot be inverted, since it would yield a positive value that is out of range and cannot be represented.

```
fixed_point<B, I, F> operator -() const
{
    fixed_point<B, I, F> result;
    result.value_ = -value_;
    return result;
}
```


Increment and Decrement

Floating point types can be incremented and decremented by 1.

```
fixed_point<B, I, F> & operator ++()
{
    value_ += power2<F>::value;
    return *this;
}

fixed_point<B, I, F> & operator --()
{
    value_ -= power2<F>::value;
    return *this;
}
```

The `boost::unit_steppable` class automatically implements `operator ++(int)` (postincrement) and `operator --(int)` (postdecrement) in terms of `operator ++` and `operator --`.

In pseudo-code notation this automatic provision of operators looks like this:

<code>boost::unit_steppable</code>
<code>operator ++()</code>
<code>operator ++(int): tmp(*this); ++tmp; return tmp;</code>
<code>operator --()</code>
<code>operator --(int): tmp(*this); --tmp; return tmp;</code>

Addition and Subtraction

Addition and subtraction are implemented for `fixed_point<B, I, F>` with `operator +=` and `operator -=`.

```
fixed_point<B, I, F> & operator +=(fixed_point<B, I, F> const& summand)
{
    value_ += summand.value_;
    return *this;
}

fixed_point<B, I, F> & operator -=(fixed_point<B, I, F> const& diminuend)
{
    value_ -= diminuend.value_;
    return *this;
}
```

The `boost::ordered_field_operators` class automatically implements `operator +` and `operator -` in terms of `operator +=` and `operator -=`.

In pseudo-code notation this automatic provision of operators looks like this:

<code>boost::ordered_field_operators</code>
<code>operator +=(s)</code>
<code>operator +(s): tmp(*this); tmp += s; return tmp;</code>
<code>operator -=(d)</code>
<code>operator -(d): tmp(*this); tmp -= d; return tmp;</code>

You should be careful with addition and subtraction, since – because of their integer implementation heritage – they can overflow.

Multiplication and Division

Multiplication and division are implemented for `fixed_point<B, I, F>` with operator `*=` and operator `/=`.

```
fixed_point<B, I, F> & operator *=(fixed_point<B, I, F> const& factor)
{
    value_ = (static_cast<fpml::fixed_point<B, I, F>::promote_type<B>::type>
              (value_) * factor.value_) >> F;
    return *this;
}

fixed_point<B, I, F> & operator /=(fixed_point<B, I, F> const& divisor)
{
    value_ = (static_cast<fpml::fixed_point<B, I, F>::promote_type<B>::type>
              (value_) << F) / divisor.value_;
    return *this;
}
```

The `boost::ordered_field_operators` class automatically implements operator `*` and operator `/` in terms of operator `*=` and operator `/=`.

In pseudo-code notation this automatic provision of operators looks like this:

```
boost::ordered_field_operators
operator *=(s)
operator *(s): tmp(*this); tmp *= s; return tmp;
operator /=(d)
operator /(d): tmp(*this); tmp /= d; return tmp;
```

The implementation of the multiplication and division presents a little challenge. You probably remember: the `fixed_point<B, I, F>` class has a `value_` member of type `B`, which is an integer type.

Now consider what happens when you multiply two integers, let's say each of length 8 bits. You will get a result that needs 16 bits. The corner case is when you square the biggest representable value, so let's do this for our example:

$$a = 255 = 0xFF$$

$$a * a = 65025 = 0xFE01$$

You clearly see that in order to be able to keep every possible result of the multiplication, you will need 16 bits, that is twice the number of bits than the original factors. In other words: two factors of n bits yield a result of $2n$ bits when multiplied together.

The number of bits of an integer is a property of its type, i.e. an `unsigned char` has a length of 8 bits, an `unsigned short` has a length of 16 bit etc. Fortunately, we can do a little type manipulation with templates, in order to find the right bit sizes and types for our multiplication results.

For each type usable as a base class `B` (aka small type), we have to find a corresponding type with twice the number of bits that can be used for the result (aka big type).

Small type	Big type
signed char	signed short
unsigned char	unsigned short
signed short	signed int
unsigned short	unsigned int
signed int	signed long long
unsigned int	unsigned long long

I have provided a set of private template structs that provide the necessary information at compile time.

```
template<>
struct promote_type<signed char>
{
    typedef signed short type;
};

template<>
struct promote_type<unsigned char>
{
    typedef unsigned short type;
};

template<>
struct promote_type<signed short>
{
    typedef signed int type;
};

template<>
struct promote_type<unsigned short>
{
    typedef unsigned int type;
};

template<>
struct promote_type<signed int>
{
    typedef signed long long type;
};

template<>
struct promote_type<unsigned int>
{
    typedef unsigned long long type;
};
```

This bigger result is only used temporarily. Fixing the decimal point after the multiplication is done with a right shift of the result of exactly F bits. After the shift, the result is cast back to the original type. And here is where you have to be careful, because the multiplication can overflow, in pretty much the same way as it can for integer multiplication.

The division is similar, in that a dividend with $2n$ bits is divided through a divisor with n bits to yield a result with n bits.

Left Shift and Right Shift

The left shift operator `<<` and right shift operator `>>` are not defined for floating point types.

However, `fixed_point<B, I, F>` defines them, since it can also be used to emulate an integer type,

when `F` is set to 0. In this case, `fixed_point<B, I, F>` behaves like an integer, but gives access to the elementary mathematical functions (i.e. useful if you want to calculate the square root of an integer number).

```
fixed_point<B, I, F> & operator >>=(size_t shift)
{
    value_ >>= shift;
    return *this;
}

fixed_point<B, I, F> & operator <<=(size_t shift)
{
    value_ <<= shift;
    return *this;
}
```

The `boost::shiftable` class automatically implements `operator >>` and `operator <<` in terms of `operator >>=` and `operator <<=`.

In pseudo-code notation this automatic provision of operators looks like this:

<code>boost::shiftable</code>
<code>operator >>=(n)</code>
<code>operator >>(n): tmp(*this); tmp >>= n; return tmp;</code>
<code>operator <<=(n)</code>
<code>operator <<(n): tmp(*this); tmp <<= n; return tmp;</code>

Input and Output

For convenience, stream input and output operators have been implemented. I have used conversion to and from `double` in order to implement these operators.

```
template<typename S, typename B, unsigned char I, unsigned char F>
S & operator>>(S & s, fpml::fixed_point<B, I, F> & v)
{
    double value=0.;
    s >> value;
    if (s)
        v = value;
    return s;
}
```

The input stream `S` is a template parameter. This allows you to use just about any stream, be it a file or string stream, be it an ANSI or wide character stream.

```
template<typename S, typename B, unsigned char I, unsigned char F>
S & operator<<(S & s, fpml::fixed_point<B, I, F> const& v)
{
    double value = v;
    s << value;
    return s;
}
```

Again, using template parameter `S` for the output stream allows you to use any stream, regardless whether it is a file or a string stream or whether it is an ANSI or wide stream.

The streaming operators delegate to the double type. I have not seen any big benefit to implement these operators from scratch for the `fixed_point<B, I, F>` type. Input and output is inherently slow, so it should be affordable to resort to floating point math in these cases.

Functions

With these operators, a lot of mathematical calculations can be readily carried out with the `fixed_point<B, I, F>` class, such as those needed in matrix multiplication, etc. However, I felt that the class would not be complete without the other functions that the C++ standard library provides for floating point types, such as `sqrt` or `sin` or `log` and more. While one could convert a `fixed_point<B, I, F>` value to floating point and then call the respective function from the C++ standard library, this somehow would somehow defeat the purpose of the `fixed_point<B, I, F>` class in the first place.

Therefore, I have also implemented the mathematical functions for the `fixed_point<B, I, F>` class. These functions are rather hard to implement correctly. I have gone to some length to choose correct algorithms, but I doubt that the quality is as high as that of some C++ standard library implementations.

`fabs`

The `fabs` function computes the absolute value of its argument.

```
friend fixed_point<B, I, F> fabs(fixed_point<B, I, F> x)
{
    return x < fixed_point<B, I, F>(0) ? -x : x;
}
```

`ceil`

The `ceil` function computes the smallest integral value not less than its argument.

```
friend fixed_point<B, I, F> ceil(fixed_point<B, I, F> x)
{
    fixed_point<B, I, F> result;
    result.value_ = x.value_ & ~(power2<F>::value-1);
    return result + fixed_point<B, I, F>(
        x.value_ & (power2<F>::value-1) ? 1 : 0);
}
```

`floor`

The `floor` function computes the largest integral value not greater than its argument.

```
friend fixed_point<B, I, F> floor(fixed_point<B, I, F> x)
{
    fixed_point<B, I, F> result;
    result.value_ = x.value_ & ~(power2<F>::value-1);
    return result;
}
```

fmod

The `fmod` function computes the fixed point remainder of x/y .

```
friend fixed_point<B, I, F> fmod(fixed_point<B, I, F> x, fixed_point<B, I, F> y)
{
    fixed_point<B, I, F> result;
    result.value_ = x.value_ % y.value_;
    return result;
}
```

modf

The `modf` function breaks the argument into integer and fraction parts, each of which has the same sign as the argument. It stores the integer part in the object pointed to by `ptr` and returns the signed fractional part of x/y .

```
friend fixed_point<B, I, F> modf(fixed_point<B, I, F> x, fixed_point<B, I, F> * ptr)
{
    fixed_point<B, I, F> integer;
    integer.value_ = x.value_ & ~(power2<F>::value-1);
    *ptr = x < fixed_point<B, I, F>(0) ?
        integer + fixed_point<B, I, F>(1) : integer;

    fixed_point<B, I, F> fraction;
    fraction.value_ = x.value_ & (power2<F>::value-1);

    return x < fixed_point<B, I, F>(0) ? -fraction : fraction;
}
```

exp

The function computes the exponential function of x . The algorithm uses the identity $e^{a+b} = e^a e^b$.

```
friend fixed_point<B, I, F> exp(fixed_point<B, I, F> x)
{
    fixed_point<B, I, F> a[] = {
        1.64872127070012814684865078781,
        ...,
        1.000000000046566128741615947508 };

    fixed_point<B, I, F> e(2.718281828459045);

    fixed_point<B, I, F> y(1);
    for (int i=F-1; i>=0; --i)
    {
        if (!(x.value_ & 1<<i))
            y *= a[F-i-1];
    }

    int x_int = (int)(floor(x));
    if (x_int<0)
    {
        for (int i=1; i<=-x_int; ++i)
            y /= e;
    }
    else
    {
        for (int i=1; i<=x_int; ++i)
            y *= e;
    }
}
```

```
    return y;
}
```

cos

Calculates the cosine.

The algorithm uses a MacLaurin series expansion.

First the argument is reduced to be within the range $-\pi$ to $+\pi$. Then the MacLaurin series is expanded. The argument reduction is problematic, since π cannot be represented exactly. The more rounds are reduced the less significant is the argument (every reduction round makes a slight error), to the extent that the reduced argument and the consequently the result are meaningless.

The argument reduction uses one division. The series expansion uses 3 additions and 4 multiplications.

```
friend fixed_point<B, I, F> cos(fixed_point<B, I, F> x)
{
    fixed_point<B, I, F> x_ = fmod(x, fixed_point<B, I, F>(M_PI * 2));
    if (x_ > fixed_point<B, I, F>(M_PI))
        x_ -= fixed_point<B, I, F>(M_PI * 2);

    fixed_point<B, I, F> xx = x_ * x_;

    fixed_point<B, I, F> y = - xx *
        fixed_point<B, I, F>(1. / (2 * 3 * 4 * 5 * 6));
    y += fixed_point<B, I, F>(1. / (2 * 3 * 4));
    y *= xx;
    y -= fixed_point<B, I, F>(1. / (2));
    y *= xx;
    y += fixed_point<B, I, F>(1);

    return y;
}
```

sin

Calculates the sine.

The algorithm uses a MacLaurin series expansion.

First the argument is reduced to be within the range $-\pi$ to $+\pi$. Then the MacLaurin series is expanded. The argument reduction is problematic, since π cannot be represented exactly. The more rounds are reduced the less significant is the argument (every reduction round makes a slight error), to the extent that the reduced argument and the consequently the result are meaningless.

The argument reduction uses one division. The series expansion uses 3 additions and 5 multiplications.

```
friend fixed_point<B, I, F> sin(fixed_point<B, I, F> x)
{
    fixed_point<B, I, F> x_ = fmod(x, fixed_point<B, I, F>(M_PI * 2));
    if (x_ > fixed_point<B, I, F>(M_PI))
        x_ -= fixed_point<B, I, F>(M_PI * 2);
```

```

fixed_point<B, I, F> xx = x_ * x_;

fixed_point<B, I, F> y = - xx *
    fixed_point<B, I, F>(1. / (2 * 3 * 4 * 5 * 6 * 7));
y += fixed_point<B, I, F>(1. / (2 * 3 * 4 * 5));
y *= xx;
y -= fixed_point<B, I, F>(1. / (2 * 3));
y *= xx;
y += fixed_point<B, I, F>(1);
y *= x_;

return y;
}

```

sqrt

The `sqrt` function computes the nonnegative square root of its argument.

It calculates an approximation of the square root using an integer algorithm. The algorithm is described in Wikipedia: http://en.wikipedia.org/wiki/Methods_of_computing_square_roots.

The algorithm seems to have originated in a book on programming abaci by Mr C. Woo.

The function returns the square root of the argument. If the argument is negative, the function returns 0.

```

friend fixed_point<B, I, F> sqrt(fixed_point<B, I, F> x)
{
    if (x < fixed_point<B, I, F>(0))
    {
        errno = EDOM;
        return 0;
    }

    fixed_point<B, I, F>::promote_type<B>::type op =
        static_cast<fixed_point<B, I, F>::promote_type<B>::type>(
            x.value_) << (I - 1);
    fixed_point<B, I, F>::promote_type<B>::type res = 0;
    fixed_point<B, I, F>::promote_type<B>::type one =
        (fixed_point<B, I, F>::promote_type<B>::type)1 <<
        (std::numeric_limits<fixed_point<B, I, F>::promote_type<B>
            ::type>::digits - 1);

    while (one > op)
        one >>= 2;

    while (one != 0)
    {
        if (op >= res + one)
        {
            op = op - (res + one);
            res = res + (one << 1);
        }
        res >>= 1;
        one >>= 2;
    }

    fixed_point<B, I, F> root;
    root.value_ = static_cast<B>(res);
    return root;
}

```


Traits Class `std::numeric_limits<>`

The `fixed_point<B, I, F>` class would not be complete without a specialization of the `std::numeric_limits<>` template. The `std::numeric_limits<>` template allows you to inquire information about any numeric type, such as its minimum and maximum values and much more. Traits are indispensable to write truly generic code, code that is agnostic of type and magically works for different types.

Trait	type	value
<code>has_denorm</code>	<code>const float_denorm_style</code>	<code>denorm_absent</code>
<code>has_denorm_loss</code>	<code>const bool</code>	<code>false</code>
<code>has_infinity</code>	<code>const bool</code>	<code>false</code>
<code>has_quiet_NaN</code>	<code>const bool</code>	<code>false</code>
<code>has_signaling_NaN</code>	<code>const bool</code>	<code>false</code>
<code>is_bounded</code>	<code>const bool</code>	<code>true</code>
<code>is_exact</code>	<code>const bool</code>	<code>true</code>
<code>is_iec559</code>	<code>const bool</code>	<code>false</code>
<code>is_integer</code>	<code>const bool</code>	<code>false</code>
<code>is_modulo</code>	<code>const bool</code>	<code>false</code>
<code>is_signed</code>	<code>const bool</code>	<code>numeric_limits¹</code>
<code>is_specialized</code>	<code>const bool</code>	<code>true</code>
<code>tinyness_before</code>	<code>const bool</code>	<code>False</code>
<code>traps</code>	<code>const bool</code>	<code>false</code>
<code>round_style</code>	<code>const float_round_style</code>	<code>round_toward_zero</code>
<code>digits</code>	<code>const int</code>	<code>I</code>
<code>digits10</code>	<code>const int</code>	<code>digits * ³⁰¹/1000</code>
<code>max_exponent</code>	<code>const int</code>	<code>0</code>
<code>max_exponent10</code>	<code>const int</code>	<code>0</code>
<code>min_exponent</code>	<code>const int</code>	<code>0</code>
<code>min_exponent10</code>	<code>const int</code>	<code>0</code>
<code>radix</code>	<code>const int</code>	<code>0</code>
<code>min()</code>	<code>fixed_point<B, I, F></code>	²
<code>max()</code>	<code>fixed_point<B, I, F></code>	²
<code>epsilon()</code>	<code>fixed_point<B, I, F></code>	²
<code>round_error()</code>	<code>fixed_point<B, I, F></code>	²
<code>denorm_min()</code>	<code>fixed_point<B, I, F></code>	³
<code>infinity()</code>	<code>fixed_point<B, I, F></code>	³
<code>quiet_NaN()</code>	<code>fixed_point<B, I, F></code>	³
<code>signaling_NaN()</code>	<code>fixed_point<B, I, F></code>	³

Debugging Helpers


Visual Studio supports debugger visualizers, which help the debugger to display variables nicely. Without specialized visualizers, the default display of variables of type `fixed_point<B, I, F>` is useless first, unless you do the proper conversion to floating point by hand.

¹ Depends on the base type. If the base type `B` is signed, the `fixed_point<B, I, F>` is signed as well. Otherwise, it is not signed.

² These values are calculated based on the template parameters.

³ These values are meaningless for the fixed-point type and are set to zero.

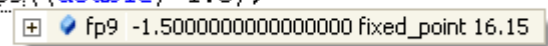
```
fpml::fixed_point<int, 16> fp9((double)-1.5);
```



The displayed value -49152 doesn't tell you anything meaningful, unless you happen to know that in order to get at the encoded value you need to divide the value -49152 by 2^{15} .

However, this is something a debugger visualizer can automatically do for you. With this visualizer, the value is displayed properly (it doesn't get the number of digits after the decimal point correct, but this is not a major issue).

```
fpml::fixed_point<int, 16> fp9((double)-1.5);
```



Visual Studio saves debugger visualizers in the text file called `autoexp.dat`, in the [Visualizer] section. This file can be found in the folder `%VSINSTALLDIR%\Common7\Packages\Debugger`. Here is the definition of the debugger visualizer for the `fixed_point<B, I, F>` type:

```
-----
; fpml::fixed_point
;-----

fpml::fixed_point<*,*,*>{
    preview (
        #if ($T3 == 32)( #($e.value_ / 4294967296., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 31)( #($e.value_ / 2147483648., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 30)( #($e.value_ / 1073741824., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 29)( #($e.value_ / 536870912., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 28)( #($e.value_ / 268435456., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 27)( #($e.value_ / 134217728., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 26)( #($e.value_ / 67108864., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 25)( #($e.value_ / 33554432., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 24)( #($e.value_ / 16777216., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 23)( #($e.value_ / 8388608., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 22)( #($e.value_ / 4194304., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 21)( #($e.value_ / 2097152., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 20)( #($e.value_ / 1048576., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 19)( #($e.value_ / 524288., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 18)( #($e.value_ / 262144., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 17)( #($e.value_ / 131072., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 16)( #($e.value_ / 65536., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 15)( #($e.value_ / 32768., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 14)( #($e.value_ / 16384., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 13)( #($e.value_ / 8192., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 12)( #($e.value_ / 4096., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 11)( #($e.value_ / 2048., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 10)( #($e.value_ / 1024., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 9)( #($e.value_ / 512., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 8)( #($e.value_ / 256., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 7)( #($e.value_ / 128., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 6)( #($e.value_ / 64., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 5)( #($e.value_ / 32., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 4)( #($e.value_ / 16., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 3)( #($e.value_ / 8., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 2)( #($e.value_ / 4., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 1)( #($e.value_ / 2., " fixed_point ", $T2,".", $T3 ))
        #elif ($T3 == 0)( #($e.value_ / 1., " fixed_point ", $T2,".", $T3 ))
    )
}
```

If you want to learn more about debugger visualizers for native code, there is a nice documentation available at <https://svn.boost.org/trac/boost/wiki/DebuggerVisualizers>.

Usage Requirements

The `fixed_point<B, I, F>` type comes in the form of a header-only library. There is no need to link anything. Just include `fixed_point.h` where needed and things should work.

The source code requires a recent version of boost (<http://www.boost.org>) and assumes that this is installed and can be found on the include path. You may need to make sure that the boost installation can be found. Boost is used for static assertions (assertions at compile time) and concept checking, to make sure that types and values are used as intended. In addition the boost operators library is used in order to automatically provide a set of operators.

All these boost libraries are header only and do require the boost files, but do not require that you go through the lengthy process of building any boost libraries.

For now I've tested with Visual Studio 2005 and 2008, but other standard conforming compilers should work as well.

Test

Together with the code I have provided a test program that tests the class and all functions. Not all possible combinations of types, integer and fractional bit sizes are tested, but a reasonable subset of parameters is.

If your requirements vary, you can easily modify the test program to make sure that your requirements are properly tested.

Benchmarks

While the primary focus of the library is the usage on embedded systems with no floating-point hardware, I couldn't resist and did some timing measurements on a PC platform. I timed various operations and compared timings with `float` and `double` types. The results are presented as clocks per operation. The benchmark program performs a large number of operations in a loop and counts the clock cycles. The total number of clock cycles is then divided by the number of repetitions.

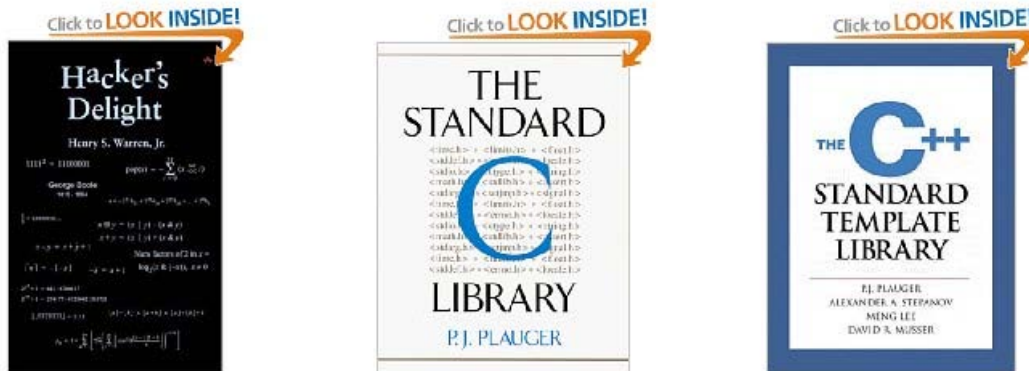
Function	float	double	15.16
Addition	1.00367	1.00365	1.00373
Multiplication	1.00368	1.00371	1.00376
Division	1.00369	1.0037	1.0037
Sqrt	1.00371	1.00371	473.881
Sine	1.00364	1.00375	162.533

You can see that the basic operations like addition, multiplication and division are fast, but the functions still need work.

Conclusion

Developing this class was quite an endeavor, and at the beginning I wouldn't have imagined how difficult it could be to write elementary functions for a fixed-point class. I have not thought about `std::numeric_limits<>` and I have not thought about debugger visualizers.

I have read a few books (a click on the thumbnail will take you to the books page on Amazon) and I have learned a lot along the way.



Finally, after a time much longer than anticipated, I got many of the things together and all in all I'm partly satisfied with the code now.

Which does not mean anything, because ultimately it is you who need to be happy as well. I'm very much interested in your feedback, which hopefully will help me and give me enough incentive to further improve the code and its usefulness. Thanks for reading thus far and good luck.