C++ BLWS (Thursday 2)

A Tour of Boost

Short breaks will be inserted as convenient.

Small helpers

1. Checked Delete
2. Identity Type
3. Base from Member
4. Uuid

Checked Delete

Boost.Checked_delete is a small helper that should be preferred over plain `delete` whenever an object of some (base) class is deleted with a virtual destructor.

> **[!]** The problem is this: built-in `delete` works with a pointer and therefore it is sufficient that only a forward declaration of a class is visible, when one of its objects is deleted. But of course, if the class had a virtual destructor, then `delete` wouldn't know about it.

With `boost::checked_delete` a syntactical context is created that fails to compile if the full class definition is not visible.

Identity Type

Boost.Identity_type allows to wrap types within round parenthesis so they can always be passed as parameters to preprocessor macros.

Base from Member

Boost.base_from_member helps to implement the C++ Base From Member idiom, to initialize a base class from some other class member.

As this is not possible with the normal initialization order, the (former) member is implemented as a (private) base class and listed left of the (public) base class that needs it for its own initialization.

Uuid

From the Boost Documentation:

> Boost.Uuid can create strings to be used as Universally Unique Identifiers in distributed environments.

Specialised Types

Tribool

Boost.Tribool implements a *three state logical* type adding `indeterminate` to `true` and `false`. Operators `&&`, `||`, and `!` `==` and `!=` are changed accordingly, so if some `boost::tribool t` that happens to be `indeterminate` is part of a (boolean) expression,

- either the outcome is still `indeterminate`, like
  - in `t || false`
  - or `t && true`,
- or it may not have any influence, like
  - in `t && false`
  - or `t || true`.

Note that whenever some `indeterminate` value is negated, its truth value is still indeterminate!

Integer

Boost.Integer provides integral types that can be picked from their properties, what makes this library particularly useful for Generic Programming.

Rational

Boost.Rational provides a template-base implementation of run-time rational numbers.

Interval Arithmetic

Boost.Interval_arithmetic applies the rules of arithmetic to intervals.

E.g. adding up two intervals gives a new interval which extends from the sum of the lower limits to th sum of the upper limits.

Multiprecision Arithmetic

From the Boost Documentation:

> The Multiprecision Library provides integer, rational and floating-point types in C++ that have more range and precision than C++'s ordinary built-in types.

Accumulators

From the Boost Documentation:

> Boost.Accumulators is both a library for incremental statistical computation as well as an extensible framework for incremental calculation in general.

Don't Reinvent the Wheel

1. Scope Exit
2. Numeric Conversions
3. Pool
4. Program Options
5. Exceptions
6. Coroutines
7. Logging
8. Test-Driven Development

Scope Exit

Boost.ScopeExit provides a small helper that allows to trigger the execution of a piece of code when the enclosing scope is exited.

In a sense, this allows to implement RAII-Style cleaning up without having to write tailored wrappers.

Numeric Conversions

From the Boost Documentation:

> The Boost.Numeric_conversion library is a collection of tools to describe and perform conversions between values of different numeric types.

Pool

From the Boost Documentation:

> Boost.Pool is a memory allocation scheme that is very fast, though limited in its usage. Pools are generally used when there is a lot of allocation and deallocation of small objects.

Program Options

From the Boost Documentation:

> Boost.Program_options allows program developers to obtain program options, that is *name-value pairs* from the user, via conventional methods such as command line and config file.

Exceptions

From the Boost Documentation:

> Boost.Exception eases the design of exception class hierarchies and helps to write exception handling and error reporting code.

Coroutines

From the Boost Documentation:

> Boost.Coroutine provides types that can be used to escape-and-reenter loops, to escape-and-reenter recursive computations and for cooperative multitasking helping to solve problems in a much simpler and more elegant way than with only a single flow of control.

Logging

From the Boost Documentation:

> Boost.Log aims to make logging significantly easier for the application developer. It provides a wide range of out-of-the-box tools along with public interfaces for extending the library. The main goals of the library are:
>
> - Simplicity. A small example code snippet should be enough to get the feel of the library and be ready to use its basic features.
> - Extensibility. A user should be able to extend functionality of the library for collecting and storing information into logs.
> - Performance. The library should have as little performance impact on the user's application as possible.

Test-Driven Development

Boost.Test is yet another C++ Unit Test Framework promising to make test-driven development easier, much like

- CppUnit,
- Google Test,
- Cute,
- or ... (you name it)

but with different strengths and shortfalls.

Design Patterns

1. Fly-Weight
2. Iterators
3. Signals2

Fly-Weight

From the Boost Documentation:

> Flyweights are small-sized handle classes granting constant access to shared common data, thus allowing for the management of large amounts of entities within reasonable memory limits. Boost.Flyweight makes it easy to use this common programming idiom by providing the class template `flyweight<T>`, which acts as a drop-in replacement for const T.

Iterators

From the Boost Documentation:

> The Boost.Iterator Library contains two parts:
>
> - A system of concepts which extend the C++ standard iterator requirements.
> - A framework of components for building iterators based on these extended concepts and includes several useful iterator adaptors.

Signals2

From the Boost Documentation:

> The Boost.Signal Library is an implementation of a managed signals and slots system. Signals represent callbacks with multiple targets, and are also called publishers or events in similar systems. Signals are connected to some set of slots, which are callback receivers (also called event targets or subscribers), which are called when the signal is "emitted."

Extended Algorithms

1. CRC
2. More String Algorithms
3. More Generic Algorithms

CRC

From the Boost Documentation:

> Boost.CRC provides two implementations of CRC (cyclic redundancy code) computation objects and two implementations of CRC computation functions. The implementations are template-based.
>
> - The first object implementation is for theoretical use. It can process single bits, but is considered slow for practical use.
> - The second object implementation is byte-oriented and uses look-up tables for fast operation. The optimized implementation should be suitable for general use.
> - The first function implementation uses the optimized object.
> - The second function implementation allows the use of a CRC that directly follows its data.

More String Algorithms

From the Boost Documentation:

> The Boost.String_algo Library provides a generic implementation of string-related algorithms which are missing in STL. It is an extension to the algorithms library of STL and it includes trimming, case conversion, predicates and find/replace functions. All of them come in different variants so it is easier to choose the best fit for a particular need.

More Generic Algorithms

From the Boost Documentation:

> Boost.Algorithm is a collection of general purpose algorithms. While Boost contains many libraries of data structures, there is no single library for general purpose algorithms. Even though the algorithms are generally useful, many tend to be thought of as "too small" for Boost.

More Containers

1. Dynamic Bitset
2. Circular Buffer
3. Heap
4. Multi-Index
5. Multi-Array
6. Intrusive Containers
7. Fusion

Dynamic Bitset

Boost.Dynamic_bitset is for those who do not like the `std::vector<bool>`.

*: Replacing the `std::vector<bool>` specialisation with `std::bitvector` seems also to be considered for C++17.

Circular Buffer

Boost.Circular_Buffer provides a fixed-size data structure suitable for queues.*

It works by advancing two pointers or indeces over the space available for contained elements, one for reading and one for writing. Presumably the read-position follows behind the write position (given filling into and taking from the data structure occur in a mix.

When the end of the available space is reached, the pointers wrap-around, reusing the slots from the front. And of course there are mechanisms to detect (and handle)

- overflow (i.w. no more space as write position is directly behind read position) and
- underflow (i.e. read position has caught-up with current write position).

*: More recent versions have been extended by an alternative data structure that can grow at runtime, if necessary.

Heap

Boost.Heap provides priority queues, which – in comparison to the `std::queue adapter` implements more functionality and different performance characteristics.

From the Boost Documentation:

> Especially, it deals with additional aspects:
>
> - **Mutability**: The priority of heap elements can be modified.
> - **Iterators**: Heaps provide iterators to iterate all elements.
> - **Mergeable**: While all heaps can be merged, some can be merged efficiently.
> - **Stability**: Heaps can be configured to be stable sorted.
> - **Comparison**: Heaps can be compared for equivalence.

Multi-Index

From the Boost Documentation:

> Boost.Multi_index enables the construction of containers maintaining one or more indices with different sorting and access semantics. It interfaces similar to STL containers. The concept is borrowed from relational database terminology and allows complex data structures in the spirit of multiply indexed relational tables where simple sets and maps are not enough.

Multi-Array

From the Boost Documentation:

> Boost.Multi_array provides a class template and semantically equivalent adaptors for arrays of contiguous data. It is a more efficient and convenient way to express N-dimensional arrays than, especially in comparison to the `std::vector<std::vector<…>>` formulation of N-dimensional arrays. Additional features, such as resizing, reshaping, and creating views are also available.

Intrusive Containers

From the Boost Documentation:

> Intrusive containers offer better performance and exception safety guarantees than non-intrusive containers (like STL containers). Their performance benefits make them ideal as a building block to efficiently construct complex containers like multi-index containers or to design high performance code like memory allocation algorithms.

Fusion

From the Boost Documentation:

> Boost.Fusion* is a library for working with heterogeneous collections of data, commonly referred to as tuples. A set of containers (vector, list, set and map) is provided, along with views that provide a transformed presentation of their underlying data.

* It is named "fusion" because it is a "fusion" of compile time meta-programing with runtime programming.

Meta Programming

Motivation

As a motivation for the benefits of Compile Time Meta Programming two other Boost libraries are presented first, a small one and a large one.

These should be only taken as examples, as many more Boost libraries exploit meta-programming techniques.

Optimized Swap

From the Boost Documentation:

> Boost.Swap provides the template function `boost::swap` that exchanges the values in two variables, using argument dependent lookup ADL to select a specialized swap function if available. If no specialized swap function is available, `std::swap` is used.

Unit-Checking

From the Boost Documentation:

> Boost.Units enables complex compile-time dimensional analysis calculations with no runtime overhead.
>
> Complete support for units and quantities (defined as a unit and associated value) for arbitrary unit system models and arbitrary value types in SI and CGS unit systems is provided, along with systems for angles measured in degrees, radians, gradians, and revolutions, and systems for temperatures measured in Kelvin, degrees Celsius and degrees Fahrenheit.

C++11: Ratio

C++11 has adopted* the Boost.Ratio library (for more information see next page).

* The C++11 <chrono> library uses compile-time rational numbers to implement durations with different resolutions, and also makes use of the (compile time) operations specified here when it has to choose the "common" type necessary for "loss-less" representation of the result, when different resolutions are combined in an operation.

Boost: Ratio

From the Boost Documentation:

> Boost.Ratio allows to specifying compile time rational constants such as 1/3 of a nanosecond or the number of inches per meter. It represents a compile time ratio of compile time constants with support for compile time arithmetic with overflow and division by zero protection.

C++11: Type Traits

C++11 has adopted much of Boost.Type_traits library (for more information see next page).

Boost: Type Traits

From the Boost Documentation:

> Boost.Type_traits contains a set of specific traits classes, each of which **encapsulates a single trait** from the C++ type system, e.g.:
>
> - Is a type a pointer or a reference type?
> - Is a type `const`-qualified?
> - Does it have a trivial constructor?
>
> Furthermore it contains a set of classes to **perform transformations** on a type, e.g:
>
> - Remove a top-level `const` or `volatile´ qualifier from a type.

Boost: Call Traits

From the Boost Documentation:

> Boost.Call_traits provide a template (to be used as compile time function) that encapsulates specifying the best method to pass a parameter of some type T to or from a function. Its main purpose and that parameters are passed in the most efficient# manner possible*

* Prior to C++11 which has changed the rules in this respect another purpose was to ensure syntax errors caused by problems like *references to references* will be circumvented.

C++11: Enable If

C++11 has adopted `boost::enable_if` (for more information see next page).

Boost: Enable If

From the Boost Documentation:

> Boost.Enable_if provides templates by which a function or class template specialization **can to include or exclude itself** from a set of matching functions or specializations, based on properties of its template arguments.
>
> E.g. one can define function templates that are only enabled for an arbitrary set of types defined by a traits class.

Boost: Preprocessor

From the Boost Documentation:

> Boost.Preprocessor provides macros to support preprocessor meta-programming. Both C++ and C compilation and the library does not depend on any other parts of Boost and therefore may be used as a standalone library.