More C++ Extensions ...

... mostly adopted from Boost

Durchführung:
Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
http://tbfe.de

Im Auftrag von:
MicroConsult
Training & Consulting GmbH
http://www.microconsult.com

Agenda

This optional section summarizes some important extensions of the C++ standard library introduced with C++11, C++14, and C++17.

- Durations, Time Points, and Clocks
- Pseudo Random Numbers (and Distributions)
- Regular Expressions
- Ad-hoc Data Structures: `std::tuple`
- Run-Time Checked Generic Pointer: `std::any`
- Single-Element Containers: `std::optional`
- Type-Tracking Unions: `std::variant`

Durations, Time Points, and Clocks

C++11 introduced a set of new classes in the sub-namespace `std::chrono` to abstract

- Time Points and
- Durations.

As for some similar libraries operators are overloaded for meaningful operations only.

E.g. you may:

- add or subtract a durations to/from a duration
- multiply or divide a duration with/by a (dimensionless) numbers
- divide a duration by a duration (yielding a dimensionless number)
- add or subtract a duration to/from a time point (yielding a time point)
- subtract a time point from a time point (yielding a duration)

But you may **not**:

- mix durations and time points in any other operation but subtraction
- mix time points and dimensionless numbers in **any** operation

Duration Example

While a generic template allows to specify a *Duration Type* in detail, most often some predefined type (based on the generic template) is used.

The following example may help to understand the most basic use:

```
namespace sc = std::chrono;
…
auto dur = sc::hours(3)
         + sc::minutes(17)
         + sc::seconds(12);
```

Here the variable `dur`

- is determined to have type* `std::chrono::seconds`
- with the (initial) value `3*3600 + 17*60 + 12` (= 4632).

Via *Overloading Suffix Operators* C++14 simplified this further (see also http://en.cppreference.com/w/cpp/symbol_index/chrono_literals):

```
using namespace std::chrono_literals;
…
auto dur = 3600h +17min + 12s;
```

*: This reveals another aspect of the elegant design: Template Meta-Programming is used in the implementation so that a combination of different duration types always will chose a result type according to the operand with the finest-grained resolution (which is seconds here).

Clock Example

Time Points often involve using a *Clock* to get the current time.

C++11 requires the following three clocks as a minimum* (see also http://en.cppreference.com/w/cpp/chrono#Clocks):

- `std::chrono::system_clock`
- `std::chrono::monotonic_clock`
- `std::chrono::high_resolution_clock`

Measuring the execution time of some piece of code could basically be done as shown here:

```
namespace sc = std::chrono;
const auto start = sc::high_resolution_clock::now();
…
… // code to time
…
const auto end = sc::high_resolution_clock::now();
using show_duration_ms = sc::duration<double, std::ratio<1, 1000>>;
std::cout << show_duration_ms(end-start) << "ms" << std:::endl;
```

*: Implementations are free to extend this, e.g. standard conforming Boost.Chrono adds clocks running only when the current process uses the CPU, allowing to measure CPU load instead of real-time.

Pseudo Random Numbers

For compatibility with C the following simple approach still works:

- generate random numbers with `rand()`,
- eventually after seeding the generator with `srand()`

The new design in C++ separated parating (random) generators from distributions, so throwing "a fair dice" requires this:*

```
int dice() {
    static std::default_random_engine random_roll{};
    static std::uniform_int_distribution<int> one_to_six{1, 6};
    return one_to_six(random_roll);
}
```

For more information see:
http://en.cppreference.com/w/cpp/numeric/random

*: While having some more lines of code it avoids the well-known deficiencies of classic much simpler approach `std::rand()%6 + 1`: the sometines very short cycle until repetition (with 100% predictability then) and the tendency to prefer the smaller values because of truncation with the modulo operator.

Regular Expressions

The following fragment shows how to break down an `std::string` assumed to hold a currency content formatted

- with full Euros into "3-digit groups" separated by dots
- and Eurocents optional, separated by a comma if present:*

```
bool euro_parser(const std::string& str, double& ec) {
    std::regex re{R"(([1-9]\d{0,2}(?:\.\d{3})*),(\d{2}))"};
    std::smatch m;
    if (!std::regex_match(str, m, re))
        return false;
    const std::string es{m[1].str()};
    ec = std::stod(std::regex_replace(es, std::regex{"\\."}, ""))
       + std::stod(m[2].str())/100.0;
    return true;
}
```

The topic is only covered further when asked for.
Then some more overview can follow based on the
Infographic "C++11 Regular Expression API" in the appendix.

*: The regular expression has a minor bug which shows for amounts of under 1€.

Ad-hoc Data Structures: `std::tuple`

The data type `std::tuple` supports to combine usually separate pieces of information into a single unit without requiring to define a specific `struct` or `class`.

A tuple can be

- constructed,
- filled,
- accessed, and
- unpacked

in several ways, furthermore ...

It is also possible to
process an `std::tuple` element by element
at compile time

... using the techniques of *Template Meta-Programming*.

Except for the Meta-Programming aspect an `std::tuple` is an extension of `std::pair` (from C++98) and there are also conversions between both.

For more information see:
http://en.cppreference.com/w/cpp/tuple)

`std::tuple` Typical Implementation

Behind the scenes `std::tuple` is implemented using compile-time recursion, but this is invisible for the users.

Generally speaking:
There is no reason why an `std::tuple` should have a

- larger memory footprint or
- worse runtime performance

compared to a `struct` holding the same data members.

In practice this means:

- there **may or may not** be substantial differences if a program replaces small "one-time use" data structures by `std::tuple`-s;*
- the memory layout of **may or may not** be binary-compatible with an equivalent `struct`.

*: The C++11 standard does not specify any guarantee so this is rather a "*Quality of Implementation*" (= QOI) Issue. The free Compiler Explorer in the Internet may be useful to find out more ...

`std::tuple` Example

Some examples for using an `std::tuple` are following. They are based on the `euro_parser` function shown before and combine two informations in the value returned:

- a `bool` indicating whether a conversion was possible
- a `double` holding the converted amount

While the modifications to the original `euro_parser` are not that substantial, it is shown once more in completion here:*

```
auto euro_parser(const std::string& str) {
    using namespace std;
    regex re{R"((0|[1-9]\d{0,2}(?:\.\d{3})*),(\d{2}))"};
    smatch m;
    if (!regex_match(str, m, re))
        return make_tuple(false, 0.0);
    const auto es{m[1].str()};
    const auto ec = stod(regex_replace(es, regex{"\\."}, ""))
                  + stod(m[2].str())/100.0;
    return make_tuple(true, ec);
}
```

*: Note that also the problem with the original regular expression has been fixed here.

`std::tuple` Example (2)

Also the new C++11 suffix return type syntax may be tried if the long-winded return type specification preceding the function name and argument list is found to be distracting:

```
auto euro_parser(const std::string& str)
    -> std::tuple<bool, double> {
    … // as before
}
```

Since C++14 the type value will be deduced from the return statement, so the function return type can be omitted if the return statements are modified to use

- either the tuple constructor explicitly ...
  - `return tuple<bool, double>{false, 0.0};`
  - `return tuple<bool, double>{true, ec};`
- ... or the factory function for tuples:
  - `return make_tuple{false, 0.0};`
  - `return make_tuple{true, ec};`

`std::tuple` Example (3)

On the receiving end the tuple must be stored and unpacked:

```
std::string input;
…
const auto result = euro_parser(input);
if (std::get<0>(result)) // branch based on success
    … std::get<1>(result) … // access amount
else
    …                       // conversion failed
```

Since C++14 members of a tuple can also be accessed by their type name (if unique inside the tuple);

```
if (std::get<bool>(result))
    … std::get<double>(result) …
else
    …
```

`std::tuple` Example (4)

Alternatively unpacking can be done with `std::tie`:

```
bool success;
double euro;
std::tie(success, euro) = euro_parser(input);
if (success) {
    … // conversion ok, amount found in `euro`
else
    … // (invalid input format)
```

If not all members of a tuple are of interest, a tie can also be done only partially, e.g. if only a validity check of the input format is desired …

```
bool success;
std::tie(success, std::ignore) = euro_parser(input);
```

… or if a zero-amount of is acceptable to indicate a failed conversion:

```
double euro;
std::tie(std::ignore, euro) = euro_parser(input);
if (euro != 0.0) … // conversion ok, amount found in `euro`
```

Single-Element Containers: `std::optional`

This class, introduced in C++17,* offers a simple standard way to add a "validity flag" to any kind of value.

> A typical use case for `std::optional` is to indicate the "*not valid*" or "*not (yet) initialized*" state for a type that uses all bit patterns of its binary representation so that there is no special value left to serve as sentinel.

E.g. a function to convert a character string into a numeric value could use an `std::optional<int>` as return value to indicate a failed conversion.

For more information see:
http://en.cppreference.com/w/cpp/utility/optional

*: Prior to this it was introduced in the C++ in 2015 with the Library Fundamentals TS as `std::experimental::optional` and before was available since long as Boost.Optional.

`std::optional` Typical Implementation

An object of type `std::optional<` *T* `>` locates its payload (*T*) in proximity with a validity flag

- on the stack,
- at a static address,
- or on the heap

just as it would be the case for any ordinary object or base type variable.

`std::optional` Example

In the `euro_parser` it could have been used as return value too:

```
auto euro_parser(const std::string& str)
    -> std::optional<double> {
    std::regex re{R("([1-9]\d{0,2}(?:\.\d{3})*),(\d{2}))"};
    std::smatch m;
    if (!std::regex_match(str, m, re))
        return {};
    const std::string es{m[1].str()};
    return std::stod(std::regex_replace(es, std::regex{"\\."}, ""))
         + std::stod(m[2].str())/100.0;
}
```

`std::optional` Example (2)

There is a default `bool`-conversion to check validity while accessing the payload will be done with the `get` member function:

```
…
if (auto result = euro_parser(input)) {
    …
    … result.get() … // access converted amount
    …
}
else {
    …                // conversion failed
}
```

More Use Cases for `std::optional`

Besides marking the validity of a returned value there are some more typical use cases:

- clearly indicate the difference between
  - variables not yet initialized and
  - variables to which a value has been assigned
- for function arguments support true "out"-parameters

To illustrate the latter, consider this:*

```
void f(const T1& in_only, T2& in_out, std::optional<T3>& out_only) {
    … in_only …; // `const` reference enforces read-only access
    … ++in_out …; // non-`const` reference allows modification
    out_only = T3{ … , … }; // assign (new) value as necessary
}
```

*: At first glance this example may seem over-engineered – shouldn't an ordinary (non-`const`) reference `T3&` suffice as `out_only` argument? Most often this may be true or make no difference, but using an `std::optional<T3>&` may still have a real advantage if there is no reasonable way to construct the receiving argument variable before. And even if there is, it may be inefficient to first construct something only to have it overwritten soon after.

Generic Pointers with Runtime-Checking): `std::any`

This class, introduced in C++17,[*]) offers an efficient replacement for generic, **un-typed** pointers (`void *`).

This is especially true if

- the type of such pointers needs to be tracked anyway, to apply
- a type-cast for accessing the pointed to object beyond the bit-level.

A typical use case for `std::any` is in layered software architecture where some "middle" layer strives to reduce dependencies.

E.g. generic communication service may want to be "agnostic" with respect to a set of types well-known at the more special data collecting, processing, and presentation layers, for which it provides the connection.

For more information see:
http://en.cppreference.com/w/cpp/utility/any

[*]: Prior to this it was introduced in the C++ in 2015 with the Library Fundamentals TS as `std::experimental::any` and before was available since long as Boost.Any.

`std::any` Typical Implementation

As the maximum payload cannot be known at compile time, `std::any` needs to use heap allocation – al least as a fall back for the general case.*

The **space overhead** beyond the payload and the access pointer is

- a type discriminator, and
- some heap management data.

(The latter is often shared between a number of heap objects and so cannot be easily attributed to a single object.)

The **runtime overhead** is that of an pointer plus some

- validity checking for retrieval, and
- some resizing heap operations for content type changes.

(Validity checks for retrieval include returning an error indication if the actual and the expected content type differ.)

---

*: The programming interface does not exclude some kind of SOO (= Small Object Optimization) being used behind the scenes, similar to what is often found in `std::string` implementations on machines with large (64 bit) word size.

`std::any` Example

```
std::any v; // `v` may later hold any (moveable or copyable) type
```

Assuming the above and everything required to be done when storing new content is hidden in an overloaded assignment operator,* then

```
v = 42;
```

... assigns an `int` ...

```
v = std::string("hello");
```

... as easily as an `std::string`.

While the default c'tor (without arguments) puts the `std::any` in an "empty" state, a value may also be assigned on construction, so ...

```
std::any v2{42};
```

... initializes a value of type `int` ...

```
std::any v3{"hello"};
```

... or type `const char *`.

---

*: The final specification according to C++17 has slightly modified and enhanced the programming interface originating from Boost.Any mainly to allow optimizations when *Moving* is cheaper than *Copying*.

`std::any` Example (2)

Furthermore the `std::any` member function

- `reset` puts it back to the "not initialised/assigned" state, and
- `has_value` may be used to test for that particular state.

The more interesting part is how content is accessed.

> On each retrieval the expected content type needs to be specified.

There are two forms which differ in their failure indication, i.e. if the current content type is different from the expressed expectation.

`std::any` Example (3)

The first technique indicates failure by throwing an exception of type `std::bad_any_cast` and applies when

- at a given point in the flow of program execution there is only one valid content type, and
- everything else would be a programming error requires fixing (or maybe some kind of soft recovery strategy).

```
try { // expecting `v` contains a `double` number
    …
    … std::any_cast<double>(v) … // retrieve and process
    …
}
catch (std::bad_any_cast& ex) { // `v` contains anything else
    …
}
```

Of course, if nothing can (or needs) to be done at that point the exception should simply be let propagate down the call stack.

`std::any` Example (4)

The second technique explicitly tests for in a series* for content types:

> Note the `any_cast` here is applied to `&v` (address of `v`) instead of plain `v`.

```
…
if (!v.has_value()) {
    … // handle the "empty" case
}
else if (auto p = std::any_cast<int>(&v)) {
    … *p … // refers to contained `int` value
}
else if (auto p = std::any_cast<double>(&v)) {
    … *p … // refers to contained `double` value
}
else if (auto p = std::any_cast<MyClass>(&v)) {
    … *p …(or)… p-> … // refers to contained `MyClass` instance
}
…
```

A final else may provide a default (without retrieval) or indicated failure.

*: The chain may be run-time optimized by ordering the tests according to probability.

Type-Tracking Unions: `std::variant`

This class, introduced in C++17,* offers an efficient replacement for overlay-types like C-style `union`-s.

This is especially true if *the currently held content-type needs to be tracked by the program logic anyway,* to select the correct member before the content can be accessed (other than just at the bit-level).*

> A typical use case for `std::variant` are containers holding a bounded set of element types as content.

The advantage over `std::any` is that the "type-aware" parts of the system can be checked for completeness at compile-time, helping to diagnose problems early if the list of types to support grows.

> (i) For more information see:
> http://en.cppreference.com/w/cpp/utility/variant

*: Prior to this it was introduced in the C++ in 2015 with the Library Fundamentals TS as `std::experimental::variant` and before was available since long as Boost.Variant.

`std::variant` Typical Implementation

As the maximum payload is known at compile time, `std::variant` can use stack or fixed-address allocation.

The **space overhead** beyond the payload is

- a type discriminator, **plus**
- the size difference between the actual and the largest payload.

(So, if there is a vast size difference of payloads and the largest one is only rarely used, `std::any` might be the better choice.)

The **runtime overhead** is that of

- some validity checking for retrieval, and
- a d'tor call for the current type when assigning a different one.*

(It is arguable whether the latter should be considered as an overhead because every instance of a class needs to be destructed at sometime.)

---

*: If such an assignment can fail the `std::variant` will not provide Strong Exception Safety but instead leave the instance in a "value-less state", which can be tested for with the `valueless_by_exception()` member function.

`std::variant` Example

The set of types that can be stored in a variant is called "bounded" because it is fixed at compile-time ...

```
std::variant<bool, float, std::string> v; // `v` may hold a `bool`
                                          //     OR a `float`
                                          //     OR an `std::string`
```

Assuming the above and everything required to be done when storing new content is hidden in an overloaded assignment operator,* then

```
v = 4.0f;
```

(assigns a `float`)

```
v = std::string("hello");
```

(assigns to `std::string`)

Due to automatic type conversions also the following works:

```
v = 4;
```

```
v = "hello";
```

`std::variant` Example (2)

But there are also c'tors accepting a single argument of each of the other types, so ...

```
decltype(v) v2{4.2};
```

... initializes it as `float` ...

```
decltype(v) v3{"hello"};
```

... or type `std::string`.

An "un-initialized state" is possible, but **not** with the default c'tor (without arguments) as this initializes an `std::variant` according to its first type.*

```
std::variant <void*, unsigned long, MyClass> x; // <== nullptr
std::variant <MyClass, void*, unsigned long> y; // <== MyClass{}
std::variant <unsigned long, MyClass, void*> z; // <== 0uL
```

[!] The un-initialized state is assumed if an assignment throws after the old content has been already cleared-out and a failure occurs before the new content is completely written.

*: Built-in types will get the same initialization as for globals.

`std::variant` Example (3)

On each retrieval the expected content type needs to be specified.

There are three forms which differ in their failure indication, i.e. if the current content type is different from the expressed expectation.

The first technique indicates failure by throwing an exception of type `std::bad_variant_cast`.*

```
try { // expecting `v` contains a `double` number
    …
    … std::get<float>(v) … // retrieve and process
    …
}
catch (std::bad_variant_access& ex) { // `v` contains anything else
    …
}
```

*: As for `std::any` this technique is usually preferable if the reason is a program bug which finally needs some fixing in the source code, re-compilation etc. The **`try – catch` may exist** and attempt to "resolve the problem" within the range of options to completely it, trying silent continuation with of without logging and some "self-healing", **or it may not exist** letting the process crash with a core file, and maybe restart it immediately or let an embedded device carry out a warm-boot …

`std::variant` Example (4)

The second technique explicitly tests for in a series* for content types:

> Note `get` here is replace with `get_if` **and**
> is applied to **&**v (address of v) instead of plain v.

```
…
if (auto p = std::get_if<bool>(&v)) {
    … *p … // refers to contained `int` value
}
else if (auto p = std::get_if<float>(&v)) {
    … *p … // refers to contained `double` value
}
else if (auto p = std::get_if<std::string>(&v)) {
    … *p …(or)… p-> … // refers to contained `std::string` instance
}
…
```

> **[!]** The problem here is that – if a new alternative is added to the variant – it gets silently ignored **or** an inappropriate default processing is chosen.

`std::variant` Example (5)

The third technique is helpful to turn "missing type cases" after a new alternative has been added into compile time errors.

This is done with a static variant of the Visitor Design Pattern. The first step is to define a Functor-like class overloading `operator()` with argument types matching the `std::variant` alternatives:*

```
struct my_variant_printer {
    void operator()(bool arg) const {
        std::cout << (arg ? 'T' : 'F');
    }
    void operator()(float arg) const {
        std::cout << arg;
    }
    void operator()(const std::string& arg) const {
        std::cout << '"' << arg << '"';
    }
};
```

*: The approach shown above was chosen to be backward compatible with Boost.Variant with only minor changes. With the extended facilities for Template Meta Programming in C++17 there are more options which may or may not be considered "even more elegant" – see here: http://en.cppreference.com/w/cpp/utility/variant/visit#Example

`std::variant` Example (6)

The Functor-like class then is

- handed over to the helper `std::visit` as first argument,
- with the variant instance to process as second argument*

```
std::visit(my_variant_printer{}, v);
```

A compile-time error will occur if the variant `v` holds alternatives which none of the overloaded `operator()` in `my_variant_printer` can handle.

---

*: With Boost.Variant **(1)** `boost::apply_visitor` is used instead of `std::visit`; **(2)** the Functor-like class has to be publicly derived from `boost::static_visitor<>`; **(3)** if some value is to be returned from the overloaded `operator()`s the type must be specified as template argument for `boost::static_visitor`.

```
// visitor-based processing of a variant (with compile-error for "forgotten" alternatives)
boost::variant<bool, …, …, MyClass> my_variant;
…
struct my_processing : boost::static_visitor<int>{
    int operator()(bool arg) const { …; …; return …; }
    …                                  // ^^^^^^^^--vvvvvvvv----- some `int`
    int operator()(const MyClass& arg) const { …; …; return …; }
};
…
int my_result = boost::apply_visitor(my_processing, my_variant);
```