C++11 BLWS (Monday 2)

Universal Basic Helpers

---

1. C++11: Callables (`std::function`)
2. Boost: Callables
3. C++11: Lambdas
4. C++ Function Objects
5. Boost: Lambdas
6. C++11: Binders (`std::bind`)
7. Boost: Binders
8. Reference Wrappers
9. C++11: Tuple
10. Boost: Tuple
11. Boost: Optional
12. Boost: Any
13. Boost: Variant

Short breaks will be inserted as convenient.

C++11: Callables (`std::function`)

C++11 adds a template to represent *Callables*.

This templates is kind of a *Compile Time Wrapper Class Generator* manufactoring objects with an overloaded function call operator.

Such objects can be initilalized with:

- Classic C Function Pointers
- C++11 Lambdas
- C++ Function Objects aka Functors*

*: The term functor is commoly applied if the main purpose of a class is to be *called* via its overloaded `operator()`. A typical example is shown when Classic C++ Function Objects are covered.

C++11: `std::function` Template Instantiaton

As `std::function` is a template*, instantiation arguments have to be supplied.

The Syntax should be obvious from the following examples:

```
std::function<void()> f1;     // no args, returning nothing
std::function<int>(int)> f2;  // int arg, returning int
std::function<double>(const char *)> f3;
std::function<double>(const char *, const char**, int)> f4;
```

Encoded in the *type* of an `std::function` are the types of its arguments and the return type.

*: Actually function objects are an application of an C++ idiom known as Type Erasure.

C++11: `std::function` Initialization / Assignment

On initialization or when assigned to a function object accepts anything

- that can be called as a function
- with the required number of arguments
- returning a value of the required type.*

*: The actual rules are a bit more complicated and allow a certain amount of type coercion to occur at the actual call. E.g. a function actually returning an `int` may be assigned to an `std::function` assuming that nothing is returned. The return value will then be discarded if the function is called through the `std::function` object.

C++11: `std::function` Calls

Calling a function through an `std::function` uses exactly the same syntax as any ordinary function call.

The following example should only demonstrate the principle, it has no real use:

```
#include <cstdlib>
#include <functional>

std::function<int(const char *)> f;

int main() {
    f = std::atoi;
    return f("42");
}
```

C++11: `std::function` Testing

Similar to a classic function pointer - **which an `std::function` isn't(!)** - it may not refer to a callable entity at some point in time:

1. If it is defined but not initialized.
2. If it is explicitely cleared.

This can be checked by simply using the `std::function` object in a boolean context:

```
#include <cassert>
#include <cstdlib>
#include <functional>

int main() {
    std::function<int(const char *)> f;
    assert(f == false);
    f = std::atoi;
    assert(f == true);
    f = nullptr;
    assert(f == false);
}
```

`std::function` Efficiency

The advanced template techniques used in the implementation of `std::function` need not be understood to just use them.

Nevertheless two key points to be kept in mind:

1. Calls through a function always involves a call tu a virtual member function, i.e. there is always the overhead of a function call implied (no inlining).
2. If the first restriction is acceptable, using an `std::function` instead of a generic type template argument may help to avoid code bloat.*

*: Depending on the compiler it may also sometimes lead to more comprehensible error messages.

`std::function` vs. Fully Generic Types (1)

A predicate in a (templated) algorithm could be restricted to a callable type via `std::function`:

```
template<typename T1, typename T2>
T2 filter(T1 beg, T1 end, T2 to,
          std::function<bool(typename T1::value_type)> pred) {
    while (beg != end) {
        if (pred(*beg))
            *to++ = *beg++;
    }
    return end;
}
```

If the above is not `inline` and if `T1` and `T2` always denote the same types, there will be only a single version of this function, **not** depending on the predicate actually used in a call.

`std::function` vs. Fully Generic Types (2)

In contrast to the previous example, with the following template the compiler is able to - **and often will*** - inline the predicate evaluation, resulting in more or less "code bloat".

```
template<typename T1, typename T2, typename T3>
T2 filter(T1 beg, T1 end, T2 to, T3 pred) {
    while (beg != end) {
        if (pred(*beg))
            *to++ = *beg++;
    }
    return end;
}
```

[!] Whether this is relevant for an application or not depends on many considerations. Therefore no general rule or guideline can be given which solution to prefer over the other.

*: Whether the actual predicate evaluation will be inlined also depends on other factors, like its definition is visible (and maybe explictely made `inline`), or its a classic function for which at the call site only an `extern` declaration is known. Also note that different compilers may very in handling the above and the final result may depend on the optimization level requested.

Boost: Callables

C++11 `std::function` emerged from `boost::function`.

There are few differences, the most important is that boost function tried to be compatible with older compilers that did not fully support the required template syntax.

So Boost supplied its *Portable Syntax* as an alternative:

```
std::function0<void> f1;     // no args, returning nothing
std::function1<int, int> f2;  // int arg, returning int
std::function1<double>, const char *> f3;
std::function3<double, const char *, const char**, int> f4;
```

- It encoded the return type as first template argument,
- followed by the argument types (if any), and
- reflected the **number of arguments** in the class name.

C++11: Lambdas

Introducing *Lambdas* with C++11 was a major step to bring C++ at level to many other modern programming laguages, in which

- functions not only were made "*First Class Citiziens*" but
- it is also possible to specify a function body **at its point of use**,* especially as an argument to some other function call.

The general definition syntax

- starts with a capture list in square brackets,
- followed by an argument list in round parenthesis,
- followed by the function body in curly braces.

*: This is why lambdas are also known as *Function Literals*.

Lambda 101 - Definition Syntax Example

Building on the `filter` algorithm from a prior slide* the predicate could be supplied directly and clearly visible at the call site:

```
std::vector<double> data, result;

…
filter(data.begin(), data.end(), std::back_inserter(result),
       [](double e) { return (e < std::sqrt(2.0)); }
};
```

Note that the above will work with either possibility, the predicate parametrized to any type or limited to an appropriate `std::function`!

*: This algorithm is also available as `std::copy_if` in C++11. (It was missing from C++98 but `std::replace_copy_if` could be used with the predicate logic inverted.)

Lambda 101 - Capture Lists (Motivation)

It should be understood that for any function (say: `filter`) expecting some other function as argument,

- the function handed over **at the call-site** (of `filter`)
- must **exactly match** the call that is coded (inside `foo`).

Therefore it is not possible handing over additional arguments directly:

```
filter(data.begin(), data.end(), std::back_inserter(result),
       [](double e, double max) { return (e < max); }
};
```

The above will **not compile** because the lambda handed over violates the expectation `filter` has about the use of its fourth argument.*

*: How the error manifests in a compiler diagnostic is a different issue: if - after some type decuction - `filter` had an argument of type `std::function<bool(double)` the compiler will complain at the call-site not beeing able to initialize the predicate argument from what is specified; for a fully generic template it will probably complain **inside** the code of the template where the actual call happens.

Lambda 101 - Capture Lists (Example)

In a capture list variables from the local context may be named.

Then in the code generated the lambda then that argument is handed through via a special path*

```
double max;
std::cin >> max;
…
filter(data.begin(), data.end(), std::back_inserter(result),
       [max](double e) { return (e < max); }
};
```

As covered so far this presentation only tried to give some first clues about the purpose and use of lambda capture lists. There are many more details which have not been covered so far. Please any appropriate C++11 reference documentation instead.

*: If you are curious about that special path you will get an idea how it works when Classic C++ Function Objects are covered.

Lambda 101 - Be Aware of the Pitfalls

Due to efficieny reasons C++11 did **not require any special rules for stack unwinding** when a lambda captures a local context by rerefence.*

```
function<void()> foo(int n) {
    return [&n]() { std::cout << n; }
}
…
foo(42)();
```

[!] Therefore the above code fragment steps into the area of undefined behavior.

- Of course, the situation might go unnoticed for a long time as the correct value just happens to be in the expected memory location ...
- **... until, some day, a completely unrelated change is made ...**

*: It is basically a similar situation as returning the address of a local from a function, which is undefined behavior since the first days of C. But while most any decent compiler will warn about this, the problem shown here most often still goes unnoticed. It can be expected that code like the above will also trigger a warning when future C++ compilers improve their checks in this respect.

Classic C++ Function Objects

Prior to C++11 lambdas - in C++98 and C++03 - code that can now can be written elegantly with lambdas had to be written with *functors*:*

```
struct LessCompare {
    const double limit;
    LessCompare(double lim) : limit(lim) {}
    bool operator()(double e) { return (e < limit); }
};
…
double max;
std::cin >> max;
…
filter(data.begin(), data.end(), std::back_inserter(result),
       LessCompare(max)
);
```

*: To avoid unnecessary complexity the `LessCompare` functor was not written as a template here. Of course this could have been easily done (and in practice probably would have done) to make the functor applicable to any type supporting `operator<()`

Boost: Lambdas

As lambdas were missing for a long time from the C++ language proper, it was tried to emulate them via the library.

- Expression templates appeared to be a promising solution.
- To rewrite the example that has been used a number of times:

```
#include <boost/lambda/lambda.hpp>
…
filter(data.begin(), data.end(), std::back_inserter(result),
       boost::lambda::_1 < max);
```

The innocuous looking `boost::lambda::_1` above together with a cleverly overloaded `operator<` triggers a complex, template based machinery.

Last and finally the result is a functor that overloads `operator()` to be callable with a single argument, returning `true` if the argument is less than `max`.

Boost: Lambda Details

While C++11 lambdas now provide a much more general and flexible solution, it can be argued that Boost lambdas are less prominent, i.e. in many simple cases they need much fewer key strokes.*

> This is especially true if the namespace `boost::lambda` is opened via a using directive, because then simply using the plain indentifiers `_1`, `_2`, or `_3` in an expression is sufficient to trigger the mechanism.

```
std::vector<double> data;
…
sort(data.begin(), data.end(), (_2 < _1)); // sort data in reverse
```

The above example, rewritten C++11 lambdas nicely illustrates the point:

```
sort(data.begin(), data.end(), [](double lhs, double rhs) {
                                   return (rhs < lhs);
                               }); // sort data in reverse
```

*: Whether or not this is considered to be an issue to guide a decission pro or contra Boost lambdas may also depend on the capabilities of an IDE, which - if properly configured - might insert a source code templates for frequently required C++11 lambda with keyboard shortcuts.

Boost: Lambda Beyond Trivial Use

Beyond the trivial use pure expressions - and even then sometimes - Boost lambdas quickly get intricate.

While it is well possible to translate the following into a Boost lambda, you probably will not want to do it if it can be avoided.*

```
vector<double> data;
…
int line{0};
std::for_each(data.begin(), data.end(),
              [&line](double e) {
                  std::cout << ++line << ':' << e << '\n';
                  if (line % 10 != 0) std::cout << "---\n";
              });
```

*: Of course, if you really have to back-port the above code written for C++11 to some older compiler it may be good to know that Boost lamdas stretch far beyond the simple, expression-like use cases and all the typical flow control directives are supported too.

C++11 vs. Boost Lambdas

Which one to prefer cannot be generally decided:

- For very simple expressions Boost lambdas may be considered as an alternative.
- On the other hand, as soon as the situation is a bit more involved, the correct use of Boost lambdas quickly becomes tricky at least.
- Generally speacking there is a steep learning curve beyond the trivial cases, especialy when flow control beyond pure expressions is required.

Surely it will not make much sense to rewrite all usages of Boost lambdas now that C++11 lambdas are available.*

*: Except of course, Boost lambdas are the only or one of only a few dependencies of a C++ project to Boost and there is the demand to remove all Boost dependencies.

C++11: Binders (`std::bind`)

Binders are a mechanism that may or may not* be preferred over lambdas in simple situations where

- a new function is created from an existing one
  - but either the new function needs fewer arguments (some arguments of the other function will receive fixed values)
  - or requires its arguments specified in a different order from the call (which may e.g. dicated by an algorithm to which the resulting function is given as argument)
  - or both.

*: Whether or not to use binders at all but stick purely with lambdas seems to be mainly a matter of taste.

Reducing Number of Arguments with `std::bind`

In the following the `std::bind` adapts the function `bar`, which has three arguments to be called from inside `foo` with only two arguments and the middle argument fixed to the square root of two:

```
#include <functional>
using namespace std::placeholders;
extern void bar(int, double, const char *);
…
foo(std::bind(bar, _1, std::sqrt(2.0), _2));
```

> It should be noted that the highest placeholder number used (`_2` above) determines argument count of the function returned from `std::bind`.

Also note that the placeholders can be used in their shortest form because they have been imported from their (sub-) namespace.*

*: Because of the identical names such use may collide with placeholders of Boost lambda. So special care has to be taken when `std::bind` is mixed with other libraries that uses similar placeholder names.

Changing Order of Arguments with `std::bind`

In the following the `std::bind` adapts the function `baz`, which has two arguments to be called from inside `foo` with the arguments reversed:

```
#include <functional>
using namespace std::placeholders;
extern void baz(const char *, int);
…
foo(std::bind(baz, _2, _1));
```

It is also possible to drop arguments completely:*

```
foo(std::bind(baz, _2, 42));
```

In both cases `std::bind` returns a function to be called with two arguments, determined by the placeholder `_2`.

*: Dropping the last argument would require a less obvious technique like using the comma operator to discard the begin of an expression sequence:
`foo(std::bind(baz, (_2, "hello"), _1));`

Binding Member Functions

It is also possible to use `std::bind` for objects and their member. The only thing special here is that objects are bound as ifg their were the member function's first argument*

The next two examples assume the following class:

```
class MyClass {
    …
    void f(int);
    void cf(int) const;
    int g(char) const;
};
```

To make the following examples applicable in a cookbook-style, they introduce Reference Wrappers which are more fully covered only later.

*: In other words: more or less binding simply makes explicit what is happening anyway ("behind the scenes") whenever a member function is called …

Binding Member Functions to Same Object

It is now possible to bind the same object to various arguments:*

```
MyClass mc;
std::vector<int> v;
std::for_each(v.begin(), v.end(),
     std::bind(&MyClass::cf, mc, _1));
```

As `std::bind` uses value arguments the following may help to improve performance ...

```
std::for_each(v.begin(), v.end(),
     std::bind(&MyClass::cf, std::cref(mc), _1));
```

... or even be necessary to make modifications visible:

```
std::for_each(v.begin(), v.end(),
     std::bind(&MyClass::f, std::ref(mc), _1));
```

---

*: Note that `std::for_each` was chosen here only because it constitutes a very simple example. Especially given range-based loops in C++11 the above could well be replaced with:
`for (auto e : v) mc.cf(e);`

Binding Member Functions to Different Objects

Also various objects may be bound to call the same member function*

```
std::vector<MyClass> v;
… // fill v
auto n = std::accumulate(v.begin(), v.end(), 0,
                         std::bind(&MyClass::g, _1, 'a'));
…
std::list<int> result;
std::transform(v.begin(), v.end(),
                          std::back_inserter(result),
                          std::bind(&MyClass::g, _1, 'z'));
```

Rewritten to a range based loops the above is equivalent to:

```
int n{0}; for (const auto &e : v) n += e.g('a');
…
for (const auto &e : v) result.push_back(e.g('z'));
```

*: It should be understood that these examples are different from the ones on the last slide as `_1` now refers to the object from the container and triggers generating the appropriate expression template, so a reference wrapper would be misplaced here.

Binding Member Different Objects to Differently

Generally speaking, possible limitations are not so much imposed by `std::bind` as by what algorithms exist.*

As there exist a variant of `std::transform` processing two containers in parallel, a selected member function may also be applied to different objects with different arguments:

```
std::string s;
… // fill s;
assert(s.size() >= v.size());
std::transform(v.begin(), v.end(), s.begin(),
               std::back_inserter(result),
               std::bind(&MyClass::g, _1, _2));
```

Translated into a range-base loop the above is equivalent to:

```
auto z{s.cbegin()};
for (const auto &e : v) result.push_back(e.g(*z++));
```

*: Not to say: The possibly limited knowledge of developers with respect to the algorithms available in C++11 and Boost … well, that's not you as you're here now :-)

`std::bind` vs. C++11 Lambdas

Everything that can be achieved with `std::bind` can be achieved with lambdas too.

The mose important difference is that now the argument types are more explicit:*

```
foo([](int arg1, const char *arg2) { bar(arg2, std::sqrt(2), arg1); }
foo([](int arg1, const char *arg2) { baz(arg2, arg1); }
foo([](int     , const char *arg2) { baz(arg2, 42); }
foo([](int arg1, const char *    ) { baz("hello", arg1); }
```

*: The similarity would become even more visible the arguments of the lambda were named _1 and _2.

Boost: Binders

C++11 `std::bind` emerged from `boost::bind`.

There are few differences most of which are related to lifted restrictions because C++11 supports true variadic templates.

- With `std::bind` the maximum number of arguments of **the adapted function** is essentially unlimited, while
- the maximum number of arguments for the function returned from `std::bind` is determined by the highest placeholder number defined in an implementaion (and therefor implementation defined).

For `boost::bind` there is a fixed limit* for both, the number of arguments of the adapted function and the number of arguments of the function returned.

*: Prior to the installation of Boost a configuration step may be run in which such configuration values can be changed - typically at the price of header file size and hence longer compilation time (with negative impacts probably only for unreasonably large limits).

Reference Wrappers

At many places in C++ prefers handling (copies of) values.

- Often this is safer and relieves the developer from the burden of always caring about possible aliasing or memory ownership.
- Where necessary references instead of a values can be explicitely chosen.
- In case of library functions the interface somtimes cannot be changed directly, but reference wrappers may be inserted instead.

Reference Wrapper Example (1)

The following function `print` automatically inserts line breaks if otherwise a given length would be exceeded:

```
void print(std::string word, unsigned maxlen, unsigned &filled) {
    if (filled + word.length() > maxlen) {
        std::cout << '\n';
        filled = 0;
    }
    if (filled > 0) {
        std::cout << ' ';
        ++filled;
    }
    std::cout << word;
    filled += word.length();
}
```

Reference Wrapper Example (2)

Assuming there are a number of words in (of type `std::string`) in any sequential container `text`, the above function could be adapted to be used with `std::for_each`:

```
unsigned line_filled = 0;
auto p80 = std::bind(print, 80, line_filled);
std::for_each(text.begin(), text.end(), p80);
```

A first text shows that it works as expected. But it fails if the text comes in two containers `text` and `moretext` where the second should continue in the last line of the first, if it's only partially filled:

```
std::for_each(moretext.begin(), moretext.end(), p80);
```

[!] The reason is that `std::bind` takes its arguments by value, so the reference to `filled` which is modified actually isn't `line_filled`.

Reference Wrapper Example (3)

The solution is to connect `line_filled` through a reference wrapper.

This is most easily done by the helper function `std::ref`:*

```
auto p80 = std::bind(print, 80, std::ref(line_filled));
```

There is also a *constant reference* version `std::cref`, providing a solution for constant but non-copyable types that need to be bound:

```
class MyClass {
    …
    MyClass(const MyClass &) = delete;
    …
};
…
MyClass obj;
void foo(…, const MyClass &arg, …);
… std::bind(foo, …, obj, …) …            // COMPILE ERROR
… std::bind(foo, …, std::cref(obj), …) … // OK
```

C++11: Tuple

Tuples are kind of ad-hoc structures and are mainly used to temporarily combine several unrelated values to be handled as unit, e.g. in a function result.

The similarity of `std::tuple` to `std::pair` for the above use case is obvious and if tuples had been part of C++98 they would have probably been used in some places were currently `std::pair` is used.

Because the number of elements in a tuple is unlimited,* their can be no names (like `first` and `second` in a pair) and a different technique must be applied to access the individual parts of a tuple.

*: Interrestingly, the other border case are not tuples with a single element, but completely empty ones. These not only may have some use in very generic, templated data structures but can also help to write compile time algorithms processing all elements of a tuple. This can be done by recursive calls of a variadic templates, terminated with a specialization for the (degenerate) empty tuple.

Definition and Initialization of `std::tuple`

When defining a tuple the type of its elements may be explcitely defined:

```
std::tuple<int, double, const char *> t;
```

This may - of course - be combined with an initialization:*

```
std::tuple<int, double, const char *> t{42, std::sqrt(2), "hello"};
```

Types can be omitted by using `auto`:

```
auto t = std::make_tuple(42, std::sqrt(2), "hello");
// or:
auto t(std::make_tuple(42, std::sqrt<float>(2), "hello"));
```

But - as a subtlety of "uniform initialization" - the following creates a tuple too, though quite a different one:

```
auto t{std::make_tuple(42, std::sqrt(2), "hello")};
```

---

*: And of course any old initialization syntax may be used here in place of the curly braces style.

Element Access of `std::tuple`

To access individual elements a global getter-function must be used:

```
… std::get<0>(t) … // the int
… std::get<1>(t) … // the double
… std::get<2>(t) … // the const char *
```

[!] It should be understood that the "element index" for `std::get` has to be a compile time constant.

Therefore the following will not work:*

```
for (int i = 0; i < 3; ++i) std::cout << std::get<i>(t) << '\n';
```

*: There are ways to print all the elements of a tuple - but it must be done with a "loop unrolling itself" completely at compile time, as for each element a different overload of `operator<<` might have to be used.

Unpacking an `std::tuple`

There is a different technique that unpacks all elements of a tuple at once:

```
int count;
double value;
const char *name;
std::tie(count, value, name) = t;
```

Obviously this has its greatest advantage if all values have to go into a variable of their own, or at least most values:

```
int count;
std::string name;
std::tie(count, std::ignore, name) = t;
```

Compared to element access with `std::get` it is said that a small overhead may be imposed by using `std::tie`, but if any it, is far from substantial.

Modifying `std::tuple`

As `std::get` returns a reference, it is straight forward to modify individual elements of a tuple:

```
std::get<0>(t) = 12;
std::get<1>(t) *= 2;
```

Tuples can be also modified as a whole, only given each of their elements are assignment compatible:

```
std::tuple<int, double> t1{3}, t2{12, std::sqrt(2)};
auto t3 = std::make_tuple(true, 0u);
t1 = t2; t2 = t3; t3 = t1;
```

Assigning all elements at once fails if (some or all) are `const`-qualified:

```
std::tuple<const int, double> t4{-1, 0.0};
const std::tuple <int, double> t5{0, 0.0};
t1 = t4; // OK (of course can assign FROM const)
t4 = t1; // FAILS because of const first member
t5 = t1; // FAILS because all members are const
std::get<1>(t4) = std::get<1>(t1); // OK
```

Boost: Tuple

C++11 `std::tuple` emerged from `boost::tuple`.

There are few differences most of which are related to lifted restrictions because C++11 supports true variadic templates.

- With `std::tuple` there is no upper limit to the number of contained elements.
- For `boost::tuple` the maximum is limited - usually to 10 or 20, depending on the configuration step run as first thing when the Boost installation.

Another difference is that `std::tuple` can **only** be accessed with the global getter while `boost::tuple` also has a member for that purpose. In the following fragment `t` is assumed to be a tuple with at least three elements:

```
… std::get<2>(t) …  // C++11 and Boost
… t.get<2>() …      // Boost only
```

Boost: Optional

The use of `boost::optional` may be considered as an alternative to using pointers and having the `nullptr` represent the case: does not exist.

- Differently from the pointer approach with `boost::optional` no heap allocation will occur.
  - Instead `boost::optional` reserves space for its payload data type and a flag (e.g. on the stack in case of a local variable) ...
  - ... using *Placement New* when the payload space is to be initialized ...
  - ... or a direct destructor call when it is made invalid.

```
boost::optional<int> x; // default initialized ...
assert(!x);             // ... it's not yet valid ...
x = 42;                 // ... now gets assigned ...
assert(x);              // ... tested here ...
assert(x.get() == 42);  // ... retrieved here
```

Boost: Any

The use of `boost::any` may be considered as an alternative to using untyped pointers (`void *`) to achieve "runtime polymorphism" for types that cannot related by a common base class.

- Internally `boost::any` refers to the assigne data via a `void*` member ...
- ... but with an additional member tracks the type assigned last ...
- ... for which it can be queried with a convenient syntax.

```
boost::any x;                // default initialized to nullptr
if (…)                       // depending on a runtime condition ...
    x = true;                // ... may actually hold a boolen ..
else if (…)                  // ... or in a different case ...
    x = std::sqrt(2.0);      // ... may actually hold a double ..
else if (…)                  // ... or yet differenly ...
    x = std::string("hi!"); // ... an std::string ...
else                         // ... or ...
    x = …;                   // ... (who knows?)
```

Boost: Any (cont.)

To access a `boost::any` its content type must always be tested for.

There is a rather systematic way to do so:*

```
if (bool *p = boost::any_cast<bool*>(x)) {
    … *p …
}
else if (double *p = boost::any_cast<double*>(x)) {
    … *p …
}
else if (std::string *p = boost::any_cast<std::string*>(x)) {
    … *p …
}
```

> The pointers `p` can also be `auto` or `auto *`-typed.

That way there is **only one** type to adapt when a code block gets copy-pasted to implement a new type case branch.

*: Here a lesser known feature from C++98 is at work, allowing to define a variable inside an `if`-condition so that it gets local scope, limited to the statement or block following, much like it is often used for counting variables in `for` loops.

Boost: Variant

The type `boost::variant` is similar to `boost::any` with the important exception that the set of types must be known at compile time.*

```
boost::variant<bool,
               std::string,
               double> x;  // from a (default initialized) bool
…                          // (as bool is the first on the list)
x = std::sqrt(2.0);        // can be set to any other type from
…                          // the list, either exactly matching
x = std::string("hello"); // but also applying conversions, if
…                          // an unambiguous choice can be made,
x = "world";               // like const char * to std::string
x = nullptr;               // or nullptr_t to bool (-> false)
```

Therefore it can be used as replacement for a classic C-style `union` augmented with a type tracking member, that is automatically set on initialization and assignment.

*: Effectively this means that there is a closer coupling to the client code as for `boost::any`, where the set of types involved of course is also fixed in the source code, but more remotely, i.e. only in the context of initialization, assignment, and value access.

Boost: Variant (cont.)

A variant can be accessed in several ways:

- Specifying the expected type directly, e.g. with `boost::get<bool>(x)`, with an exception thrown when the content is different.
- By trying a type with `auto *p = boost::get<bool>(&t)`, followed by an explicit or implicit test of `p` against the `nullptr`.*
- With a compile time variant of the visitor pattern.

```
struct process : boost::static_visitor<> {
    void operator()(bool b) const { … } // process content if bool
    void operator()(double d) const { … }         // ... if double
    void operator()(const std::string &s) const { … }  // ... etc.
};
…
boost::apply_visitor(process(), x);
```

Note that applying a visitor detects missing cases at compile time!

*: In so far similar to `boost::any` but with slightly different syntax.

Boost: Variant (cont.)

In vistor-style there is also the option to fold cases with common source code into a template, while still keeping special cases special:*

```
struct print : boost::static_visitor<> {
    std::ostream &os;
    print(std::ostream &os_) : os(os_) {
        os.setf(ios::boolalpha);
    }
    template<typename T> void operator(const T &v) const {
        os << v;
    }
    void operator(const std::string &v) const {
        os << '"' << v << '"';
    }
};
…
boost::apply_visitor(print(std::cout), x);
```

Missing type cases now might not have the required implementation.

*: For `boost::variant` an overloaded `operator<<` is already defined though without any special-casing.