C++11 BLWS (Monday 2)

Universal Basic Helpers

---

1. C++11: Callables (`std::function`)
2. Boost: Callables
3. C++11: Lambdas
4. Function Objects
5. Boost: Lambdas
6. C++11: Binders (`std::bind`)
7. Boost: Binders
8. C++11: 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 template 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
- Function Objects aka Functors*

*: The term *Functor* is commonly applied if the main purpose of a class instance is to be *called like a function* (via its overloaded `operator()`). A typical example is shown when classic C++ function objects are covered.

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

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

The syntax should be obvious from the following examples:

```
std::function<void()> f1;      // no args, returns nothing

std::function<int(int)> f2;        // int arg, returns int

                                        // const char* arg,
std::function<double(const char *)> f3;  // returns double

                     // const char*, const char**, int arg,
                                           // returns double
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 a C++ idiom known as Type Erasure.

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

On initialization, or when something is assigned to it, an instance of type `std::function` 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 certain type conversions to occur at the actual call.

*: A function, functor, or lambda actually returning a value may be assigned to an `std::function` instance that assumes a callable to return nothing. The result will simply be dropped from the assigned callable when a call is made through the function object not returning a value.

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

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

Note that the following example only demonstrates 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");
}
```

*: In case you are eager to demonstrate that `f` has actually been called here, with Unix or Linux this is not difficult: instead of running the executable with `./a.out` use `echo $(./a.out)`.

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

Like a function pointer* it **may or may not** refer to a callable entity:

1. When in its initialization no Concrete Callable was provided.
2. After it has been explicitly cleared, typically by assigning a `nullptr`.

The state can be checked by using the instance in a boolean context:

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

int main() {
    std::function<int(const char *)> f;
    assert(!f);    // using boolean negation from classic C
    f = std::atoi;
    assert(f);
    f = nullptr;
    assert(not f); // using alternative form of boolean negation
}
```

*: It should be understood that despite the similarities an `std::function` object **is not a function pointer** and especially requires a more heavy-weight implementation.

`std::function` Efficiency

The implementation of `std::function` uses the Type Erasure Idiom* and can just be used in "cookbook-style" if two key points are kept in mind:

1. Calls through an `std::function` involve calling a virtual member function. There is typically the overhead of an indirect function call at assembler level – **inlining is usually not possible**.
2. If the first restriction is acceptable, preferring an `std::function` over a generic type may reduce code bloat for templated predicates.

---

*: Type erasure need not be understood in detail but works (approximately) along the following lines:

```
// modelled here for the SPECIAL case function<int(bool, const char *)> ONLY:
struct Fwd { virtual Fwd *clone() =0; virtual int call(bool, const char *) =0; virtual ~Fwd() {} };
template<typename ActualImplementor> struct SpecificForwarder : public Fwd {
    ActualImplementor implementor;
    SpecificForwarder(const ActualImplementor &impl) : implementor(impl) {}
    virtual Fwd *clone() { return new ActualImplementor(implementor); }
    virtual int call(bool b, const char *s) { return implementor(b, s); }
};
struct TypeErased {
    std::unique_ptr<Fwd> fwd;
    template<typename T> explicit TypeErased(const T &impl) : fwd(new T(impl)) {}
    TypeErased(const TypeErased &other) : fwd(other.fwd ? other.fwd->clone() : nullptr) {}
    int operator()(bool b, const char *s) { return fwd->call(b, s); }
    … // more ops not shown (assignment, test for empty etc.)
};
```

Using `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) {
        const auto &val = *beg;
        if (pred(val))
            *to++ = val;
        ++beg;
    }
    return to;
}
```

If `T1` and `T2` denote types (for `beg`/`end` or `to` respectively)

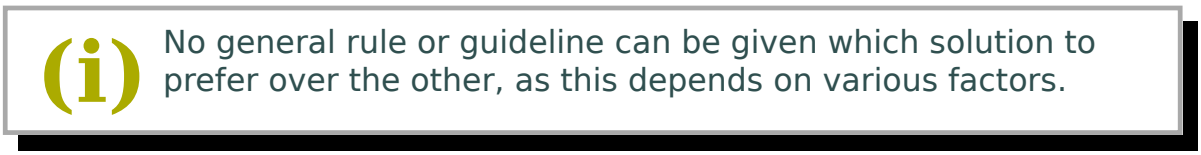
- which do **not vary** for many calls
- that all use **different** predicates `pred`

there will be **only one single** instantiation of this function. (I.e. the instantiation will not depend on the actual predicate.)

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

In contrast to the previous example, with the following template the compiler is able* to inline the predicate evaluation, resulting in some "code bloat":

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



*: Whether the actual predicate evaluation will be inlined also depends on other factors, like its definition is visible (and maybe explicitly made `inline`), or whether it is a classic function for which at the call site only an `extern` declaration is known.

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 *Preferred Syntax*.

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

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

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

---

*: With the current state of compiler support the portable syntax is obsolete now, maybe except for some few exotic platforms with limited support for syntax features required since C++98.

C++11: Lambdas

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

- functions not only were made *"First Class Citizens"* but
- it is also possible to specify a function body **at its point of use**,* especially as 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;
… // fill data
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 version of filter, the one with the predicate parametrized to any type and the one with the predicate limited to an appropriate callable.

*: An algorithm like this is available as `std::copy_if` in C++11. (It was actually missing from C++98 where `std::remove_copy_if` had to be used with the predicate logic inverted.)

Lambda 101 – Capture Lists (Motivation)

It should be understood that for any function (e.g. `filter`) expecting some other function as argument,

- the function handed over by the caller
  - i.e. as **argument to** `filter`
- must be callable by the code inside
  - i.e. the **implementation of** `filter`.

Therefore it is not possible to hand over additional arguments directly:

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

[!] This will not compile because the lambda not matches 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 deduction – `filter` had an argument of type `std::function<bool(double)>` the compiler will complain at the call-site not being able to initialize the predicate argument from what is specified; for a fully generic template it will rather complain inside the code of the template where the actual call is coded.

Lambda 101 – Capture Lists (Example)

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

In the code generated captured variables are transfered via a special path:*

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

So far this presentation only tried to give some first clues about the purpose and basic use of lambda capture lists.

> There are many more details which have not been covered yet, like handing over references in the capture list or some shortcuts for it.

Please lookup more information in the relevant reference documentation.

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

Lambda 101 – Beware of the Pitfalls

For efficiency reasons C++11 does **not demand any special rules for stack unwinding** when a lambda captures a local context by reference:*

```
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 variable 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 goes unnoticed. It can be expected that code like above will trigger a warning too 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 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; // only as example (max not known at compile-time)
…
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.

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 clever overload of `operator<` – triggers a complex template based machinery.*
- Last and finally a functor is created overloading `operator()` to be callable with a single argument, returning `true` if the argument is less than `max`.

*: If you are curious to learn the details try to get familiar with Expression Templates first, because these build the underlying general mechanism.

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 blatant and in many cases of practical relevance can be create with fewer key strokes.*

> This is especially true if the namespace `boost::lambda` is opened via `using` directives: then the occurrence of the plain identifiers `_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 for C++11 lambdas 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 decision pro or contra Boost.Lambdas may also depend on the capabilities of an IDE, which – if properly configured – might insert source code templates for frequently used C++11 lambdas with a keyboard shortcuts.

Boost: Lambda Beyond Trivial Use

Beyond the trivial use in pure expressions – and even there sometimes – using Boost Lambdas (or rather the expression templates behind them) can quickly get intricate.

While it is well possible to translate the following into a Boost.Lambda, you probably will not want to do it when you can avoid it:*

```
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.Lambdas stretch far beyond the simple, expression-like use cases and all the typical flow control directives are supported.

C++11 lambdas vs. Boost

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, correct use of Boost Lambdas quickly becomes tricky (at least).

Generally speaking with Boost Lambdas there is a steep learning curve beyond the trivial cases, especially when flow control beyond pure expressions is required.

> It will not make much sense to rewrite all usages of Boost Lambdas now that C++11 lambdas are available, but rewriting the complicated ones to better comprehensible C++11 lambdas may be worth a consideration*

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

Binders as defined in the C++11 standard 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** and
  - either the new function needs fewer arguments because some arguments of the existing function will receive fixed values,
  - or requires its arguments specified in a different order from a given call (which may e.g. be dictated by an algorithm to which the resulting function is handed over as argument),
  - or both.

C++11 Lambdas can do all of the above too – and more!*

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

Reducing Number of Arguments with `std::bind`

In the following `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));
```

Note that the placeholders can be used unqualified here because they have been imported from their (sub-) namespace.*

The highest placeholder number used (`_2` above) determines the (lower bound of the) argument count of the function returned from `std::bind`.

*: Because of the identical names making placeholders visible as `_1`, `_2` etc. may collide with Boost.Lambda placeholders or Boost.Bind placeholders. In general special care must be taken when `std::bind` is mixed with other libraries that use such placeholder names too for their own purpose.

Changing Order of Arguments with `std::bind`

In the following `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 that needs to be called with two arguments, determined by the placeholder `_2`.*

*: More exactly, the placeholder `_X` with the largest value of `X` determines that (at least) a number of `X` arguments are required to call the function returned from `std::bind`. Supplying lesser arguments in a call will result in a compilation error.

Binding Member Functions

It is also possible to use `std::bind` for objects and their members – shown here only for member functions but it works the same for any accessible member data.

The only thing special here is that objects are bound as if they 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 possible to bind the same object to various arguments:*

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

As `std::bind` uses value arguments the use of a *reference wrapper* like `std::cref` may help to improve performance …

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

… or `std::ref` may even be necessary to make modifications visible:

```
std::for_each(data.begin(), data.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 : data) mc.cf(e);`

Binding Member Functions to Different Objects

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

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

Rewritten to range based loops the above is equivalent to:

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

*: Note that these examples are different from the ones on the previous slide as `_1` now refers to the object from the container and triggers generating the appropriate expression template, so no reference wrapper is required here.

Binding Member of Different Objects with Different Arguments

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

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

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

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

```
auto z = data.cbegin();
for (auto &e : objs) 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 most visible difference is that with lambdas the argument types are more explicit:

```
foo([](int _1, const char *_2) { bar(_2, std::sqrt(2), _1); });
foo([](int _1, const char *_2) { baz(_2, _1);               });
foo([](int   , const char *_2) { baz(_2, 42);               });
foo([](int _1, const char *  ) { baz("hello", _1);          });
```

> The above example emphasizes this similarity by using `_1` and `_2` as name for the formal lambda arguments.

Usually such names were rather chosen according to their purpose, like:

```
foo([](int count, const char *name) { baz(name, count); });
```

*: With C++11 lambdas as well as with Boost lambdas though the topic will only be discussed for the former here.

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 implementation.

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

*: Whether there is a fixed limit depends on the compiler used and what exactly is the limit is determined be a configuration step that is run prior to the installation of Boost. Increasing the default limit typically has a price in header file size and hence longer compilation time (with negative impacts probably only for unreasonably large limits).

C++11: Reference Wrappers

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

- Often this is safer and relieves the developer from the burden of caring about aliasing issues or memory ownership.
- Where necessary, references instead of a values can be explicitly chosen.
- In case of library functions the interface sometimes cannot be changed easily, here reference wrappers may be inserted instead.

Reference wrappers have already been used in two prior examples for `std::bind`:

- One such use (`std::cref`) was only to improve performance,*
- without the other (`std::ref`) the code wouldn't have worked as intended.

---

*: Not all uses of `std::cref` are only to improve performance – for classes which make their objects non-copyable they may be necessary so that the code compiles in the first place.

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 of type `std::string` in a 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 test shows that it works as expected. But it fails if the words come in two containers `text` and `moretext` where the second should continue printing 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` but a copy of it in the generated (invisible) binder object.

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
```

*: Of course reference wrappers are not a recent invention of C++11 but had been available via Boost.Ref since long … (only, due to the close similarity to C++11 they do not receive a slide of their own here).

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* they cannot be names (like `first` and `second` in a pair) and a different technique must be applied to access the individual parts of a tuple.

*: Interestingly, 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 to process all elements of a tuple. This can be done by recursive calls of 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 explicitly 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 classic 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` a small overhead may be imposed by `std::tie`, but surely far from substantial.*

*: Vague language is used here to indicate that details will vary between implementations and future compilers may apply code-generation techniques to improve chances for optimization, putting `std::tie` at par with direct member assignment.

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 first member is const
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 (optionally) run as first thing during 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:*

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

*: Of course, here `t` is assumed to be a tuple with at least three elements.

Boost: Optional

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

- Differently from the pointer approach with `boost::optional` no heap allocation will occur.
  - Instead `boost::optional` reserves space for its payload data type **plus** a flag (e.g. on the stack in case of a local variable) ...
  - ... uses Placement `new` to eventually initialize the payload ...
  - ... and an Explicit Destructor Call when it is to be invalidated.

```
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
```

*: `std::experimental::optional` is part of the C++ library fundamentals TS and can be expected to be merged in C++17 as `std::optional`.

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 be related through a common base class.

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

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

---

*: `std::experimental::any` is part of the C++ library fundamentals TS and can be expected to be merged in C++17 as `std::any`.

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 … // access member if of type bool
}
else if (double *p = boost::any_cast<double>(&x)) {
    … *p … // access member if of type double
}
else if (std::string *p = boost::any_cast<std::string>(&x)) {
    … *p … // access member if of type std::string
}
```

> 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)
```

Compared to `boost::any` this means there is a closer coupling to the client but also more compile time type checking becomes possible.

> You may view `boost::variant` as improved C-style `unions`.

*: `boost::variant` can be expected to become part of C++17 as `std::variant`.

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>(&x)`, followed by an explicit or implicit test 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);
```

Applying a visitor detects missing cases at compile time!

*: In so far it is 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 handling 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);
```

But: Missing cases **now** might not have the required implementation!