C++11 BLWS (Monday 1)

Some Fundamentals

Short breaks will be inserted as convenient.

C++11 – Syntax Cleanups

An important syntax cleanup is to allow to write adjacent *less-than* signs as closing angle brackets for template argument lists:*

```
map<string, vector<int>> wordposlist;
```

While some compilers (like Visual Studio) since long handled the above according to the C++11 rules, others (like GCC) required this:

```
map<string, vector<int> > wordposlist;
```

Some extra-careful developers even wrote:

```
map<string, vector<int>/**/> wordposlist;
```

*: It should be noted that this change breaks previously valid code that used the right-shift operator in an expression to specify a template value argument. Therefore `g++` changes its behaviour depending on the option `-std=c++11` (or `-std=c++0x` for less recent versions). Old code with shift operators as part of a template value argument must put the expression in parentheses.

C++11: auto-typed Variables

The old C keyword `auto` has changed its meaning with C++11:

- It now specifies that the type for some variable is deduced from its initializing expression.
- Note that type modifiers may be added but also might be stripped from the initializing expression.*

```
auto x = 3;          // x has type int
auto y = 0uL;        // y has type unsigned long
auto p1 = &x;        // p1 has type int*
auto *p2 = &x;       // p2 has type int* too (!)
const auto cx = 42;  // cx has type const int BUT ...
auto ncx = cx;       // ... ncx has type int (NOT const int)
int &ri = x;         // ri has type int& BUT ...
auto nri = x;        // ... nri has type int (NOT int&)
```

*: The rules are very close to the rules applied when for template functions types are deduced from actual arguments.

C++11: Builtin `decltype`

The compiler-builtin `decltype` is now available to

- determine the type of a variable or
- an expression (which will not be evaluated!)

```
// continuing the example from the previous page
… decltype(x) …              // represents type int
… decltype(ri) …             // represents type int&
… decltype(x+y) …            // represents type unsigned long
… decltype(std::sqrt(-1)) …  // represents type double
```

> One main use for `decltype` is in templates to determine the result of an operation with operands of a dependant type.*

For more information see:
http://en.cppreference.com/w/cpp/language/decltype

*: If the type is necessary for the result of a function the new suffix return type syntax of C++11 comes in handy.

Boost: Type-of

See also BoostBook: typeof

Also Boost.Typeof provides extensions similar to `auto` and `decltype`.

As these had to be implemented as library functionality* they

- are much more limited and clumsy to use,
- result in less readable code.

> Therefore it can be expected that such parts of Boost become obsolete as soon as C++11 is implemented by all compilers relevant for productive use with a software project's code base.

*: Of course, where C++98 compilers provided useful non-standard extensions the Boost functions made use thereof.

C++11: Uniform Initialization

C++ traditionally had many forms of initialization, some of which were limited to certain contexts:

```
int x = 0;                         // traditional style
const std::string greet("hi"); // constructor style
struct s {
    int a;
    char z;
} v = { 42, '!' };                 // aggregate initialization
std::string empty();               // INVALID (as initialization)
unsigned u = unsigned();           // not common but valid (in C++)
```

Since C++11 curly braces may be used in any initialization context:*

```
int x{0};                          // explicit zero initialization
const std::string greet{"hi"}; // initialization by constructor
string empty{};                    // valid for default constructor
unsigned u{};                      // implicit zero initialization
```

*: Compared to classic initialization some rules have slightly changed: E.g. if the value of the initializing expression cannot be represented in the initialized variable, this is a compile time error.

C++11: Initializer Lists

Initializer lists are sequences of comma-separated values enclosed in curly braces.

- They are valid wherever a function accepts an argument of type `std::initializer_list`.
- This includes many constructors for standard containers:*

> **[!]** A few usage forms introduced ambiguities for which C++11 defined disambiguating rules – sometimes little intuitive ones.

```
vector<short> primes({ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 });
const map<string, string> words = {
    { "zero", "null" }, { "one", "eins" }, { "two", "zwei" },
    …
};
vector<int> x{3};  // A vector initialized with a "list" just
                   // holding a single 3?
                   // Or rather a vector sized to 3 elements
                   // that shall be default-initialized?
```

Boost: Value Initialized

See also BoostLibs: utility/value_init

Using the correct initialization syntax can pose a problem* at times:

```
template<typename T> void foo() {
    T local = … // ???
    …
}
```

- A plain `T local;` would default initialize classes but leaves basic types uninitialized.
- Classic style `T local = 0;` would only zero-initialize basic types.*

Boost.Value_initialized provides a utility template, causing either zero or default initialization, depending on the type of T:

```
value_initialized<T> local; // uniform use of the Boost solution
T local = T();              // a (somewhat lesser known) C++98 solution
T local{};                  // C++11 uniform initialization as solution
```

* Actually the state of affairs is a bit more complicated: The above would also work if T were a class with a (non-explicit) constructor taking an argument type to which 0 can be converted. Besides all arithmetic types this include bool (0 converted to false) and any pointer type (0 converted to nullptr).

Boost: Container Initialization

See also BoostLibs: assign/doc/index

Boost.Assign provides some operator overloading to allow a more readable initialization syntax for sequential and associative containers.

Overloaded `operator,` and `operator+=` help with sequential containers:

```
vector<int> primes;
primes += 2, 3, 5, 7, 11, 13, 17, 19, 23, 29;
```

For associative containers there is a somewhat tricky overload of `operator()` (function call):

```
map<string, string> words;
words(("one")("eins"))
     (("two")("zwei"))
     …
     (("nine")("neun"))
     ;
```

Compared to C++11 initializer lists the above not only looks clumsy but also has the draw-back that no `const`-qualifiers are possible.

C++11: Range-Based `for` Loops

C++11 supports a new and uniform syntax to loop over all elements in a collection:

```
vector<int> primes;
…
for (auto v : primes)
    … // access element through v
```

Nothing changes if `primes` were any other sequential or associative container*, a built-in array, or even an `std::initializer_list`:

```
for (auto v : { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 })
    … // access element through v
```

> It is only little effort – if there is no standard iterator interface anyway – to make user-supplied containers iterable with range based loops.

*: For maps the placeholder variable will be an `std::pair` to access the key via `v.first` and the associated value via `v.second`.

Boost: Foreach

See also BoostBook: foreach

Like some other third party libraries* Boost.Foreach tries to outwit C++98 with a macro – called `BOOST_FOREACH` in this case – that mimics what has become a built-in in C++11:

```
vector<int> primes;
…
BOOST_FOREACH(int v, primes)
    … // access elements via v
```

*: An assembler code analysis of `BOOST_FOREACH` is still pending on behalf of the author of this text. But it was done for the Qt version once, and in that case the result was a convincing argument **against** using such trickery … at least when code efficiency is the primary target: For built-in types Qt's for-each produced as much as ten times the amount of code compared to a classic `for` loop!

C++11: Move Semantics

Move semantics provide the solution to two problems that could not (always) be avoided in C++98:

- Efficient use of *value types as function return values*, e.g. if they represent large containers.
- Implementing types that are *movable* but not *copyable*.*

Even before C++11 in many practical cases the leeway given to a compiler to apply RVO and NRVO could achieve much to **return large data structures by value efficiently**.

But as there is no guarantee in this respect, the usual recommendation for C++98 was to hand-out large containers via reference arguments, not by return value – and this recommendation should possibly be followed even in C++11.

---

*: With C++98 there is no real solution to the problem to differentiate between moveable and copyable types. Even if `operator=` stays undefined and overloaded global functions `assign` and `move` were used consequently, especially in type-generic code of template implementations something in the vein of Perfect Forwarding could not be achieved, at least not with as little code as can now.

C++11: Rvalue References

Move semantics heavily build on Rvalue References defined with a double ampersand:

```
void foo(const T &classic_reference) { … } //first
void foo(T &&rvalue_reference) { … } //second
```

With the above two overloads C++11 would bind

- the first `foo` to arguments that are plain variables (including `const` qualified variables),
- the second `foo` to arguments that are temporaries which will be destroyed soon after use.*

```
T a; const T b; extern T bar();
foo(a);        // calls first
foo(b);        // calls first
foo(bar()); // calls second
foo(a+a);   // calls second -- provided T supports operator+
```

*: Such as function calls, but also including constants and expressions (with some reasonable exemptions).

Copyable and Movable Types

Instances of the following class will be both, copyable and movable:*

```
class MyClass {
    …
public:
    MyClass(const MyClass &); // classic copy constructor and ...
    const MyClass& operator=(const MyClass &); // copy assignment
    …
    MyClass(MyClass &&);        // C++11 move constructor and ...
    const MyClass& operator=(MyClass &&);      // move assignment
};
```

By supplying both of move and copy support, or only the one or the other, or none at all, instances of `MyClass` can easily be made

- **Copyable** and **Movable**,
- **Copyable** but not Moveable,
- not Copyable but **Movable**, or
- neither Copyable nor Moveable.

*: Whether it makes sense to return the assigned-to object as a `const` or a non-`const` reference is discussed controversly with pro's and con's on both sides.

C++11: `default`-ed and `delete`-d Operations

C++11 furthermore provides a particular syntax to request or forbid compiler generated constructors and assignments, making it easy to write a class that supports the required behavior:*

```
class MyClass {
    …
public:
    MyClass(const MyClass &)                      = delete;
    const MyClass& operator=(const MyClass &)  = delete;
    …
    MyClass(      MyClass &&)                     = default;
    const MyClass& operator=(      MyClass &&) = default;
 };
```

It is probably easy to understand that instances of the above class will be **moveable but not copyable** and what needs to be changed if that should be changed.

*: In case the default implementation provided – which is member-wise move (and member-wise copy if it were also `default`-ed) is inappropriate a specific implementation can be supplied as usual.

Boost: Noncopyable

See also BoostBook: core

As C++ always generates a copy constructor when none is specified,* the usual technique is to *declare-but-not-implement* the unwanted operation.

Via deriving from `boost::noncopyable` the intent can be made more obvious (and code a bit more compact):

```
class MyClass : boost::noncopyable {
    …
    … // whatever (but no need any more to define
    … // operations that never get implemented)
    …
};
```

*: Note that with C++11 the rules changed in so far as **no default copy-constructor will be provided if a move-constructor is provided**, and the same holds for copy- and move-assignment. The reasoning behind that rule is that as soon as a specific behavior is necessary for one, copy or move, it will probably also be the case for the other.

C++11: Static Assertions

With C++11 `static_assert` arbitrary compile time checks can be expressed to abort a compilation. They are sometimes useful inside templates and available for

- code blocks (mixed with code to execute at runtime) and
- class definitions (outside executable code blocks).

```
template<typename T, size N>
class RingBuffer {
    static_assert(N < 1000, "unreasonable large size");
    …
};
```

Since C++17 the second argument is optional.

> For more complicated tests static assertions may want to move the actual calculations to a `constexpr` Function.

Boost: Static Assert

See also BoostBook: static_assert

Static assertions are also available from Boost via

- `BOOST_STATIC_ASSERT`

but as prior to C++11 such assertions had somehow to "be turned into ordinary syntax errors" the error messages finally issued tended to be less indicative of the problem or even slightly misleading.*

*: A common technique was to turn the assertion into the definition of an array with a zero or negative size, which is illegal in C and C++.

C++11: constexpr Functions

If a function is marked with the new `constexpr` keyword it will – given it adheres to certain limitations – be "compiled inside the compiler" and hence be callable at compile time.

Besides possible performance improvements for functions called with compile-time constants as arguments anyway, the result of such functions can be used in any context that requires a compile-time constant.*

```
constexpr bool is_pwrof2(unsigned v) {
    return v != 0 && (v == 1 || (!(v & 1) && is_pwrof2(v >> 1)));
}
…
const unsigned N = 4095; // should always be some 2^n - 1
static_assert(!is_pwrof2(N+1), "N is not some 2^n - 1");
```

*: With the cases of practical importance being array dimensions, template value arguments, static assertions, or in turn arguments to call other `constexpr` functions.

Compile-Time Calculations with Templates

Prior to C++11 compile-time calculations had to be carried out using meta-programming techniques based on templates:*

```
template<unsigned long long v>
struct is_pwrof2 {
    static const bool result = !(v & 1)
                                && is_pwrof2<(v >> 1)>::result;
};
template<> struct is_pwrof2<0uLL> {
    static const bool result = false;
};
template<> struct is_pwrof2<1uLL> {
    static const bool result = true;
};
```

The "call syntax" for such a function* – when `N` is another compile-time constant – will be: … `is_pwrof2<N>::result` …

---

*: Differently from `constexpr` functions, which will be also available with a run-time version to be called with non-constant arguments, a function implemented with meta-programming techniques via a template could of course not be called in a loop for testing purposes!

Support for Meta-Programming in Boost

Though `constexpr` functions provide a nice alternative to template-based techniques when non-trivial calculations have to be carried out, there are numerous other applications of meta-programming with templates.

In such cases there will most probably be the need

- to "*operate on types*" (single ones or lists thereof)
- as "*input to*" and "output from" a metaprogram.*

Boost.Hana, Boost.MPL, and Boost.Fusion are a *Libraries for Meta-Programming* with STL-like containers and algorithms – besides other useful things.

---

*: As Meta-Programming with Templates is a large and demanding topic that alone could fill some days in a course like this, it will not be covered any further (except on special demand). In general meta-programs type calculations are dominant and only occasionally mixed with value-based calculations. Nevertheless "meta-programmer" may sometimes have to *cross the compile-time to run-time border*. Besides suppurt for the latter Meta-Programming Libraries provide containers capabable of storing types and algorithms to handle their content compile time.

C++11: Template Typedefs

Despite the name* this new C++11 syntax is not limited to templates – instead it can fully replace the classic `typedef` syntax:

```
// old style typedef-syntax:
typedef unsigned long METER;
typedef void *pointer_type;
typedef const char *(*CV)(int);
```

```
// new style using-syntax:
using METER = unsigned long;
using pointer_type = void*;
using CV = const char* (int);
```

Finally the motivating (and name-giving) example:

```
template<typename CharType, std::size_t AllocSize>
class basic_fstring {
    …
};
template<std::size_t N> using fstring = basic_fstring<char, N>;
template<std::size_t N> using wfstring = basic_fstring<wchar_t, N>;
```

*: The name is a relict from the motivation finally leading to the syntax shown here, which provided a more general solution than originally was aimed for.