C++ Eleventy for ATS

Reprise with lessons learned

Introduction

- C++11 (“eleventy”) is a major change in the language (Lisp rulz!)
- 7.0 release requires compilers that are C++11 compliant.
- Take advantage of new language features without going crazy.
  - You know who you are.
- Focus here is on ATS related features, not an overview of all changes.
  - Looking at just the features I think are the most useful for ATS coding.
  - Based on experience since the last summit, these are things I have used.
- But really, it’s because Leif needs constant reinforcement.

Basics

- Use nullptr and not NULL.
- Use auto but don't go crazy with it.
- Lambdas!
- static_assert is good.
- Use C++ style casts.
- Methods should have 0 or 1 of { virtual, override, final }
  - Base method should have virtual and overrides should marked override.
- Use inline methods when possible instead of #define.
- Use std::unique_ptr for local resources. Cleanup can be overridden.

Range-based for loops

- Iterate over *elements* in a container.

```
for (auto& item : acl_list) {
  item.update();
  // …
}
```

- Container must support C++ standard iteration via
  - `begin` / `end` methods
  - `begin` / `end` functions
  - An array with a declared length.
- The variable for elements can be a value which will be a copy of the actual element.
- The variable does not have to be declared auto but that's easier.

Range based for loops

- Note that without a reference type **the iteration element is a copy**.
  - This is fine for cheap or irrelevant copies (e.g. pointers) but not for objects.
- begin and end can be free functions in order to allow range iteration on types that don't have these methods.
- C arrays are a special case to enable them to be used in this context.
- Not always usable – primary case is where an index or iterator should be returned (e.g. find).
  - See if using an STL algorithm would suffice instead.
- Will try to update some internal containers (e.g. DLL<>) to support this.

Declaration Initialization

- Class members can be initialized directly in the class declaration.
  class Bob { int x = 1; };
- Overridden by constructor member initialization.
- Avoid unobvious repetition in multiple constructors.
- Much simpler to add members with correct initialization.
  - And no more “X initialized before Y” compile time errors!

Explicit conversion operators

- User type conversions can be marked `explicit`.
- This allows direct conversions but not implicit conversions.
- Classic example – conversion to `bool`.
  - In C++98 this also allows further conversion to pointer or numeric types.
  - Worked around by making the conversion to a pointer to method.
  - In C++11 mark conversion `explicit`.

Uniform initialization

- Bit of a misnomer but allows a wide variety of initializations with the same syntax.
  - Aggregate initialization.
  - Constructor invocation.
    - Uniform initialization can never be mistaken for a function definition.
    - This is preferred if the class has a constructor.
- POD without a constructor is considered an aggregate.
  - For PODs with few members very handy. Pass temporary without writing a trivial constructor.
  - Downside – must know the order of the argument types.
- Notation is recursive.

Uniform initialization

- Can be overridden with constructor.
  - Single argument of type std::initializer_list<T>.
  - Note the list must be homogenous.
- Can be used in container loops.

  for (auto const& item : { “one”, “two”, “three” }) { … }
  - This can avoid having to declare an array used in only one loop.

Uniform initialization - arrays

- Array elements are initialized in the same order as the initialization list.
- Elements not specified are default initialized.
- An empty initialization list disables any initialization.
  int v[10] = {}; // uninitialized array
- A single {0} for an array of integral values zero initializes the entire array.
  Int v[10] = {0} // zero initialized array
  - First element gets the explicit ‘0’
  - Other elements are default initialized, which for integral types is ‘0’

Generated Method Control

- There are a number of methods that can be generated by the compiler.
  - Default constructor, destructor copy constructor, move constructor, copy assignment, move assignment
- Whether a method is generated is deterministic but the rules are a bit obscure.
- Eleventy enables direct control via the default and delete keywords attached to the method declarations. default forces the compiler to generate the method and delete forces the compiler to not generate it.
- Example: Force the default constructor for class Bob – Bob() = default;
  - Modeled on pure virtual method definitions.

Applying Eleventy – an example

Fixing up AclRecord

Current code.

Let’s add declaration initialization and clean up the constructors.

```
struct AclRecord {
  uint32_t _method_mask;
  int _src_line;
  typedef std::set<std::string> MethodSet;
  MethodSet _nonstandard_methods;
  bool _deny_nonstandard_methods;
  static const uint32_t ALL_METHOD_MASK = ~0; //
Mask for all methods.

AclRecord() : _method_mask(0), _src_line(0),
_deny_nonstandard_methods(false) {}
  AclRecord(uint32_t method_mask) :
_method_mask(method_mask), _src_line(0),
_deny_nonstandard_methods(false) {}
  AclRecord(uint32_t method_mask, int ln, const
MethodSet &nonstandard_methods, bool
deny_nonstandard)
    : _method_mask(method_mask),
      _src_line(ln),
      _nonstandard_methods(nonstandard_methods),
      _deny_nonstandard_methods(deny_nonstandard)
  {
  }
```

Constructor cleanup

The default constructor is now empty. The compiler can be given greater scope for optimization if it generates the code.

```
struct AclRecord {
  uint32_t _method_mask = 0;
  int _src_line = 0;
  typedef std::set<std::string> MethodSet;
  MethodSet _nonstandard_methods;
  bool _deny_nonstandard_methods = false;
  static const uint32_t ALL_METHOD_MASK = ~0; //
Mask for all methods.

AclRecord() {}
  AclRecord(uint32_t method_mask) :
_method_mask(method_mask) {}
  AclRecord(uint32_t method_mask, int ln, const
MethodSet &nonstandard_methods, bool
deny_nonstandard)
    : _method_mask(method_mask),
      _src_line(ln),
      _nonstandard_methods(nonstandard_methods),
      _deny_nonstandard_methods(deny_nonstandard)
  {
  }
```

No copying!

In the context in which the ACL records are constructed, the std::set used to initialize the record is used **only** to initialize the record. Therefore rather than making a copy the container should be moved. At the same time to prevent unexpected copies the copy constructor and copy assignment operator can be removed.

```
struct AclRecord {
  uint32_t _method_mask = 0;
  int _src_line = 0;
  typedef std::set<std::string> MethodSet;
  MethodSet _nonstandard_methods;
  bool _deny_nonstandard_methods = false;
  static const uint32_t ALL_METHOD_MASK = ~0; //
Mask for all methods.

  AclRecord() = default;
  AclRecord(uint32_t method_mask) :
_method_mask(method_mask) {}
  AclRecord(uint32_t method_mask, int ln, MethodSet
&&nonstandard_methods, bool deny_nonstandard)
    : _method_mask(method_mask),
      _src_line(ln),
      _nonstandard_methods(nonstandard_methods),
      _deny_nonstandard_methods(deny_nonstandard)
  {
  }
```

Provide moveable argument

Unfortunately this doesn't compile because a named local does not match an r-value reference parameter. The compiler must be specifically informed this is OK by use of std::move.

```
// IPAllow.cc:223
{
AclRecord::MethodSet
nonstandard_methods;
// …
acls.push_back(AclRecord(acl_method_m
ask, line_num,
std::move(nonstandard_methods),
deny_nonstandard_methods));
```

Switch to vector

Still not quite there because push_back uses the copy constructor to copy the argument in to the container. The code doesn't compile for that reason. Vec, the current container, isn't adequate. Instead std::vector will be used to provide the emplace_back method. This method directly constructs the instance in the container using constructor arguments via perfect forwarding. std::vector requires a move constructor which is easily added by asking the compiler to do it, as all the members are simple types or STL containers. The default constructor can be dropped as well as std::vector does not need it.

```
// IPAllow.h
 AclRecord(uint32_t method_mask) :
_method_mask(method_mask) {}
  AclRecord(uint32_t method_mask, int
ln, MethodSet &&nonstandard_methods,
bool deny_nonstandard)
    : _method_mask(method_mask),
      _src_line(ln),
_nonstandard_methods(nonstandard_met
hods),
_deny_nonstandard_methods(deny_nonst
andard)
  {}
  // Prevent copying.
  AclRecord(AclRecord const&) = delete;
  AclRecord& operator = (AclRecord
const&) = delete;
  // Move constructor for std::vector
  AclRecord(AclRecord&&) = default;
```

```
acls.emplace_back(
  acl_method_mask,
  line_num,
  std::move(nonstandard_methods),
  deny_nonstandard_methods
);
```

IPAllow.cc changes

Using emplace_back to add to the container.

Example notes

- Reduce copies of complex objects.
  - Try to remove the copy constructor, see what happens.
  - Use move semantics where possible. Always check if that's reasonable.
  - Use the emplace methods to avoid a copy.
- Prefer compiler generated methods. This makes it easier for the compiler to be clever.

Variadic Templates

Variadic Templates

- The C++11 approach to variable arguments – “var-args”.
- Somewhat more complex in exchange for complete type safety.
- Simple cases are usable by non-experts.

Variadic Templates

- The template function declares a family of functions with varying number and type of arguments.
  template < typename … Args > zwoop(Args… args) { /* code */ }
- Such a declaration is instantiated as a different function for every distinct set of arguments.
- As with var_args there can be fixed leading arguments.
  Template < typename A, typename B, typename … Rest >
  zwoop(A a, B b, Rest … rest) { /* code */ }
- As before a distinct function is instantiated for every distinct set of arguments, with the requirement the first two must be of type A and B.
- Important fact – the variadic argument can contain zero elements.
  - E.g. for the latter zwoop(a, b) matches the template.

Accessing variadic arguments

- Most common technique – currying.

```
template < typename Car, typename … Cdr > void zwoop(Car car, Cdr … cdr) {
  do_something(car);
  zwoop(cdr…);
}
```

- Each instantiation handles the first argument then passes on the rest.
  - But this doesn't actually compile – "no matching function for call to **zwoop()**"
  - Must add base case for tail recursion **before** the template so it is defined inside the template context.

```
void zwoop() {}
```

Variadic expansion is compile time

- Therefore…what? Well, this doesn't work

```
template < typename Car, typename ... Cdr > void zwoop(Car car, Cdr ... cdr)
{
  std::cout << "Arg" << std::endl;
  if (size...(cdr) > 0) zwoop(cdr...);
}
```

- Why? Because even in the case where cdr is empty, the call to zwoop() still exists even if not called and that won't compile.

Ur-Example of Variadic Template use

- printf like function with type safety.

```
void tsp() { }
template < typename Car, typename ... Cdr > void tsp(Car car, Cdr ... cdr) {
  std::cout << car;
  tsp(cdr...);
}
```

Another example – emplace() method

- Many STL containers take advantage of variadic arguments and perfect forwarding* to improve performance.
- emplace enables a caller to directly construct an element of a container instead of creating and copying.
- The method is variadic so it can take any number and type of arguments and pass them exactly to the element constructor.
- Any valid constructor can be used and will be selected by the argument set, just as if the constructor were called directly.

* Ask me after if you want “perfect forwarding” details explained.

Iterative variadic use

- For the daring, arrays and tuples can be used to capture the arguments.
- Tuples don’t seem useful, I couldn’t see a good technique in this context.
- Arrays
  - Types must be unified, generally by calling a function on the arguments.
  - Can store results or depend on side effects.

Iterative version of type safe printing

```
#include <iostream>
template < typename ... Args > void tsp(Args const& ... args) {
  (void)(int []){ ( ( std::cout << args ) , 0 ) ... };
}
int main(int, char**) {
  int a = 56; float b = 5.6;
  tsp(a, " leif ", b);
  return 0;
}
```

Change TS_DEBUG and the like?

- Formatting syntax is hard.
- Choose format string style vs. in line style.
  - printf is format string style.
  - C++ stream I/O is in line style.
- In line seems clunkier but is much safer.

Variadic Template Examples

ex-2.cc:

```
#include <iostream>
// Compile failure example.
template < typename Car, typename ... Cdr > void zwoop(Car car, Cdr ... cdr)
{
  std::cout << "Arg" << std::endl;
  if (sizeof...(cdr) > 0) zwoop(cdr...);
}

Int main(int, char**)
{
  zwoop(1, "leif");
  return 0;
}
```

ex-1.cc:

```
#include <iostream>

void zwoop() {}
template < typename Car, typename ... Cdr > void zwoop(Car car, Cdr ... cdr)
{
  std::cout << "Arg" << std::endl;
  zwoop(cdr...);
}
Int main(int, char**)
{
 zwoop(1, "leif");
 return 0;
}
```

ex-3.cc:

```
#include <iostream>

void tsp() {
}
template < typename Car, typename ... Cdr > void tsp(Car car, Cdr ... cdr)
{
  std::cout << car;
  tsp(cdr...);
}

Int main(int, char**)
{
  tsp(1, "leif");
  return 0;
}
```

ex-4.cc:

```
#include <iostream>
// Failure example - bad arguments to tuplesize
size_t tuplesizeof() {
  return 0;
}
template < typename Car, typename ... Cdr > size_t
  tuplesizeof(Car car, Cdr ... cdr) {
  return sizeof(Car) + tuplesizeof(cdr...);
}

int
main(int, char**)
{
  tuplesizeof(1, "leif");
  tuplesizeof(std::cout, 1.0, 'c');
  return 0;
}
```

ex-8.cc:

```
#include <iostream>
// ex-4.cc fixed, make args const&
size_t tuplesizeof() {
  return 0;
}
template < typename Car, typename ... Cdr >
size_t tuplesizeof(Car const& car, Cdr const& ... cdr) {
  return sizeof(Car) + tuplesizeof(cdr...);
}

int
main(int, char**)
{
  tuplesizeof(1, "leif");
  tuplesizeof(std::cout, 1.0, 'c');
  return 0;
}
```

ex-5.cc:

```
#include <iostream>

template < typename ... Args > size_t tuplesizeof(Args const& ... args) {
  size_t total = 0;
  int r[] = { sizeof(args) ... };
  for (auto x : r) {
    std::cout << "Size = " << x << std::endl;
    total += x;
  }
  return total;
}
int main(int, char**)
{
 std::cout << "Size is " << tuplesizeof(1, "leif") << std::endl;
 std::cout << "Size is " << tuplesizeof(std::cout, 1.0, 'c') << std::endl;
 return 0;
}
```

ex-6.cc:

```
#include <iostream>

struct summation {
  size_t _sum = 0;
  template <typename T> void operator() (T const&) { _sum += sizeof(T); }
};
template < typename ... Args > size_t tuplesizeof(Args const& ... args) {
  summation s;
  (void)(int []) { (s(args) , 0) ... };
  return s._sum;
}
int main(int, char**)
{
  std::cout << "Size is " << tuplesizeof(1, "leif") << std::endl;
  std::cout << "Size is " << tuplesizeof(std::cout, 1.0, 'c') << std::endl;
  return 0;
}
```

ex-7.cc:

```
#include <iostream>

template < typename ... Args > void tsp(Args const& ... args) {
  (void)(int []){ ( ( std::cout << args ) , 0 ) ... };
}

int main(int, char**)
{
  int a = 56;
  float b = 5.6;
  tsp(a, " leif ", b);
  return 0;
}
```