More Songs About Building and Foot

Stephan Bergmann

September 2016
Where we stand with C++11/14

- Since LibreOffice 5.1: GCC 4.7, MSVC 2013
- MSVC 2015 (GCC 4.8) would bring:
  - constexpr (partial; HAVE_CXX11_CONSTEXPR)
  - ref-qualifiers (HAVE_CXX11_REF_QUALIFIER)
  - thread-safe statics (HAVE_THREADSAFE_STATICS)
  - Inheriting constructors
  - noexcept
  - Unicode string literals: `s.replaceAll(u"na\u00EFve", "clever")`
    - already for single-char literals:
      `s.replaceAll(OUStringLiteral1(0x2117), "(P)")`
  - Not much energy currently put into a baseline bump, though
C++17 Sugar

- Decomposition declarations:
  ```
  auto [it, ins] = s.insert(n);
  ```

- Initializers in if statements:
  ```
  if (auto [it, ins] = s.insert(n); !ins) {
    std::cout << *it << " already present\n";
  }
  ```

  instead of
  ```
  std::pair<std::set<int>::iterator, bool> ins = s.insert(n);
  if (!ins.second) {
    std::cout << *ins.first << " already present\n";
  }
  ```
C++17 Sugar

- **Constexpr if:**

```cpp
template<typename T> bool isNonNegative(T value) {
    if constexpr (std::is_signed<T>::value) {
        return value >= 0;
    } else {
        return true;
    }
}
```

instead of

```cpp
template<typename T> typename std::enable_if<std::is_signed<T>::value, bool>::type isNonNegative(T value) { return value >= 0; }
```

```cpp
template<typename T> typename std::enable_if<std::is_unsigned<T>::value, bool>::type isNonNegative(T value) { return true; }
```
Gerrit loplugin buildbot

Jenkins
Patch Set 1: Build Started http://ci.libreoffice.org
Jenkins
Patch Set 1: Build Started http://ci.libreoffice.org

Jenkins
Patch Set 1: Verified+1
Build Successful
http://ci.libreoffice.org/job/lo_gerrit/1050/ : SUCCESS
http://ci.libreoffice.org/job/lo_gerrit_master/20955/ : SUCCESS
You must be careful in the forest
Broken glass and rusty nails
If you’re to bring back something for us
I have bullets for sale

Tom Waits/William Burroughs
auto, revisited

What is bad about the following?

```cpp
std::map<OUString, OUString> aLabels = ...;
for (std::pair<OUString, OUString> const & rLabel: aLabels)
    ...
```
auto, revisited

What is bad about the following?

```cpp
std::map<OUString, OUString> aLabels = ...;
for (std::pair<OUString, OUString> const & rLabel: aLabels)
    ...
```

Copying, that’s what:

```cpp
std::pair<OUString const, OUString> const & tmp
    = *aLabels.begin();
std::pair<OUString, OUString> const & rLabel = tmp;
```
auto, revisited

What is bad about the following?

```cpp
std::map<OUString, OUString> aLabels = ...;
for (std::pair<OUString, OUString> const & rLabel: aLabels)
...
```

Copying, that’s what:

```cpp
std::pair<OUString const, OUString> const & tmp = *aLabels.begin();
std::pair<OUString, OUString> const & rLabel = tmp;
```

Easy fix:

```cpp
for (auto const & rLabel : aLabels)
```
What’s a transparent container?

Consider

```cpp
struct Item { OUString id; ... };  
std::map<OUString, Item> items;  
std::set<Item> items;

Item getItem(OUString const & id) {
    return items.???();
}
```
What's a transparent container?

Pre 3d97b2000979200db53f77db20e882e85c66c0b6:

```
struct Item { OUString id; ... };  
std::set<Item> items;  
static Item findItem;  
Item getItem(OUString const & id) {  
    findItem.id = id;  
    return *items.find(findItem);  
}
```
What’s a transparent container?

C++11:

```cpp
struct Item { OUString id; ... };;
std::set<Item> items;

Item getItem(OUString const & id) {
    return *std::find_if(
        items.begin(), items.end(),
        [id](Item const & i) { return i.id == id; });
}
```
What’s a transparent container?

C++14:

```cpp
struct Item { OUString id; ... };  
bool operator <(Item const &, OUString const &);  
bool operator <(OUString const &, Item const &);  

std::set<Item, std::less<> > items;  

Item getItem(OUString const & id) {  
    return *items.find(id);  
}
```
What’s a transparent container?

The innocuous little

```cpp
std::set<Item, std::less<> > items;
```

instead of the default

```cpp
std::set<Item, std::less<Item>> items;
```

marks it as a C++ *transparent container*, opting in to

```cpp
items.find(id)
```

actually compiling.
Return Value Optimization (RVO) in action:

```cpp
std::unique_ptr<FilterCache> FilterCache::clone() const {
    auto pClone = o3tl::make_unique<FilterCache>();
    pClone->m_lTypes = m_lTypes;
    pClone->m_lFilters = m_lFilters;
    ...
    return pClone;
}
```
Return Value Optimization (RVO) **not** in action:

```cpp
std::unique_ptr<FilterCache> FilterCache::clone() const {
    auto pClone = o3tl::make_unique<FilterCache>();
    pClone->m_lTypes = m_lTypes;
    pClone->m_lFilters = m_lFilters;
    ...
    return std::move(pClone);
}
```

and same for temporaries:

```cpp
return std::move(FilterCache());
```
Can a virtual function be defined as deleted?

virtual void f() = delete;
Fun trivia

Can a virtual function be defined as deleted?

    virtual void f() = delete;

    Yes!  (No vtable? At least Itanium ABI has __cxa_deleted_virtual.)
Fun trivia

Can a virtual function be defined as deleted?

```cpp
virtual void f() = delete;
```

Yes! (No vtable? At least Itanium ABI has __cxa_deleted_virtual.)

Can a pure virtual function override a non-pure one?

```cpp
struct A { virtual void f() {} };  
struct B: A { void f() override = 0; };
```
Fun trivia

Can a virtual function be defined as deleted?

```cpp
virtual void f() = delete;
```

**Yes!** (No vtable? At least Itanium ABI has `__cxa_deleted_virtual`.)

Can a pure virtual function override a non-pure one?

```cpp
struct A { virtual void f() {} };
struct B: A { void f() override = 0; };
```

**Yes!** (Get rid of `{ assert(false); /* never call this */ } implementations?)
And we know what we're knowing
But we can't say what we've seen

Talking Heads