Bio-Formats C++ Conversion
Differences between C++ and Java

Roger Leigh

Tuesday 7th October 2014
University of Dundee
Overview

Basic language differences
- Java types and classes
- C++ types and classes
- Exception handling
- Interfaces

Reference handling and memory management
- Pointer problems
- Smart pointers and RAII
- Reference usage

Containers
- Arrays
- Storing different types
- variant examples
<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td>C++ released (Cfront)</td>
</tr>
<tr>
<td>1989</td>
<td>C++ 2.0</td>
</tr>
<tr>
<td>1996</td>
<td>(Java 1.0)</td>
</tr>
<tr>
<td>1998</td>
<td>ISO C++ Standard (C++98)</td>
</tr>
<tr>
<td>2003</td>
<td>ISO standard corrections (C++03)</td>
</tr>
<tr>
<td>2007</td>
<td>ISO C++ library updates (C++TR1) [current baseline]</td>
</tr>
<tr>
<td>2011</td>
<td>ISO C++ Standard (C++11) [current standard]</td>
</tr>
<tr>
<td>2014</td>
<td>ISO standard corrections (C++14)</td>
</tr>
</tbody>
</table>
Java types

Primitive types

```java
int i; double d;
```

- No unsigned integer types

Classes

```java
Pixels pixels = new Pixels();
```

- Classes, all derived from root `Object`
- Objects are by reference only
- Objects and arrays are always allocated with `new`
- Destruction is non-deterministic
Java types

Arrays

```java
Pixels[] array = new Pixels[5];
```

- Arrays have intrinsic size
- Arrays are safe to index out of bounds (throws exception)
C++ types

**Primitive types**

```cpp
int16_t i1;
uint32_t i2;
double d;
```

- Includes unsigned integer types
- Integer types of defined sizes
C++ types

Classes

```cpp
Pixels pixels;

Pixels *pixelsptr1 = new Pixels();
const Pixels *pixelsptr2 = &pixels;

Pixels& pixelsref(pixels);
```

- Classes have no common root
- All types may be instances, pointers or references
- Pointers and references may refer to `const` type
- Pointers may be `const`
- Destruction is deterministic
- **Never ever use `new`! Seriously.**
C++ types

Arrays

- Arrays “decay” to bare pointers
- Arrays are not safe to index out of bounds
- Size information lost at runtime
- Never use arrays outside static initialisers

```cpp
Pixels array[5];
```
Simplified type names

```cpp
typedef std::vector<std::string> string_list;
string_list l;
string_list::const_iterator i = l.begin();
// NOT std::vector<std::string>::const_iterator

typedef std::vector<Pixels> plist;
plist pl(6);
plist::size_type idx = 2;
// size_type NOT unsigned int or uint32_t
pl.at(idx) = ...;
```

- Standard container types e.g. `size_type`, `value_type`
- Used widely in classes and class templates
- Consistency needed for generic programming
Exceptions

Java
- `throws` details exceptions thrown
- "checked" exceptions

C++
- Exception specifications are useless except for `nothrow`
- Exceptions can be thrown at any point
- But: **Never throw in a destructor**
- Not necessary or typical to check exceptions except where needed
- All code must be exception safe
Interfaces

Java: Single inheritance, plus interfaces

C++: Multiple inheritance

- Interfaces are classes with:
  - No instance variables
  - Pure virtual methods
  - protected default constructor
  - public virtual destructor
  - Deleted copy constructor and assignment operator

- Classes implementing interfaces:
  - Use public inheritance for parent class
  - Use virtual public inheritance for implemented interfaces
  - virtual destructor
C++ pointers: pitfalls of “dumb” pointers

Automatic allocation of values (stack)

```cpp
{
    Image image(filename);
    image.read_plane();

    // Object destroyed when i goes out of scope
}
```

- Destructor run and memory freed automatically.
C++ pointers: pitfalls of “dumb” pointers

Manual memory allocation

```cpp
{  
    Image *i = new Image(filename);

    i->read_plane();

    // Memory not freed when pointer i goes out of scope
}
```

- `new` and `delete` must always be paired
C++ pointers: pitfalls of “dumb” pointers

Manual memory allocation and deallocation

```cpp
{  
    Image *i = new Image(filename);

    i->read_plane();

    delete i;
}
```

- `new` and `delete` must always be paired
- Requires manual management of ownership and lifetime
- This isn't sufficient
C++ pointers: pitfalls of “dumb” pointers

Manual memory allocation and deallocation

```c
{
    Image *i = new Image(filename);
    i->read_plane(); // throws exception; memory leaked
    delete i; // never called
}
```

- **new** and **delete** must always be paired
- Requires manual management of ownership and lifetime
- Bare pointers are not exception-safe
- Need to clean up for every exit point in a function
C++ pointers: pitfalls of “dumb” pointers

One correct solution

```cpp
{
    Image *i = new Image(filename);

    try {
        i->read_plane(); // throws exception
    } catch (const std::runtime_error& e) {
        delete i; // clean up
        throw; // rethrow
    }
}

delete i; // never called for exceptions
```

- Painful and error prone over an entire codebase
C++ pointers: std::shared_ptr as a “smart” pointer

```c++
{ // Image *i = new Image(filename);
  std::shared_ptr<Image> i
      (std::make_shared<Image>(filename));

  i->read_plane(); // throws exception

  // Memory freed when i's destructor is run
}
```

- Memory is freed by the `shared_ptr` destructor
- `shared_ptr` object lifetime manages the resource
- May be used as class members; lifetime of class instance
- Clean up for all exit points is automatic and safe
- Allows ownership transfer and sharing
- Allows reference without ownership using `weak_ptr`
RAII: Resource Acquisition Is Initialisation

- Class is a proxy for a resource
- Resource is acquired when object is initialised
- Resource is released when object is destroyed
- Manage any resource (memory, files, locks, mutexes)
- C++ language and runtime guarantees make resource management deterministic and reliable
- Safe for use in any scope
- Exception safe
- Used throughout modern C++ libraries and applications
# C++ reference variants

<table>
<thead>
<tr>
<th></th>
<th>Non-constant</th>
<th>Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pointer</strong></td>
<td>Image *</td>
<td>const Image *</td>
</tr>
<tr>
<td></td>
<td>Image * const</td>
<td>const Image * const</td>
</tr>
<tr>
<td><strong>Reference</strong></td>
<td>Image &amp;</td>
<td>const Image &amp;</td>
</tr>
<tr>
<td><strong>Shared pointer</strong></td>
<td>std::shared_ptr&lt;Image&gt;</td>
<td>std::shared_ptr&lt;const Image&gt;</td>
</tr>
<tr>
<td></td>
<td>const std::shared_ptr&lt;Image&gt;</td>
<td>const std::shared_ptr&lt;const Image&gt;</td>
</tr>
<tr>
<td><strong>Shared pointer reference</strong></td>
<td>std::shared_ptr&lt;Image&gt; &amp;</td>
<td>std::shared_ptr&lt;const Image&gt; &amp;</td>
</tr>
<tr>
<td></td>
<td>const std::shared_ptr&lt;Image&gt; &amp;</td>
<td>const std::shared_ptr&lt;const Image&gt; &amp;</td>
</tr>
<tr>
<td><strong>Weak pointer</strong></td>
<td>std::weak_ptr&lt;Image&gt;</td>
<td>std::weak_ptr&lt;const Image&gt;</td>
</tr>
<tr>
<td></td>
<td>const std::weak_ptr&lt;Image&gt;</td>
<td>const std::weak_ptr&lt;const Image&gt;</td>
</tr>
<tr>
<td><strong>Weak pointer reference</strong></td>
<td>std::weak_ptr&lt;Image&gt; &amp;</td>
<td>std::weak_ptr&lt;const Image&gt; &amp;</td>
</tr>
<tr>
<td></td>
<td>const std::weak_ptr&lt;Image&gt; &amp;</td>
<td>const std::weak_ptr&lt;const Image&gt; &amp;</td>
</tr>
</tbody>
</table>

Java has one reference type. Here, we have 22…
C++ reference usage rationalised

Class members

```cpp
Image i; // Concrete instance
std::shared_ptr<Image> i; // Reference
std::weak_ptr<Image> i; // Weak reference
```

Arguments

```cpp
void read_plane(const Image& image); // ownership retained
// ownership shared/transferred
void read_plane(const std::shared_ptr<Image>& image);
```

Return

```cpp
Image get_image(); // ownership transferred
Image& get_image(); // ownership retained
std::shared_ptr<Image> get_image(); // ownership shared/trans
std::shared_ptr<Image>& get_image(); // ownership shared
```
Safety: References can not be `null`.

Storing polymorphic types requires use of a `shared_ptr`.

Safety: To avoid cyclic dependencies, use `weak_ptr`.

Safety: To allow object destruction while maintaining a safe reference, use `weak_ptr`.

`weak_ptr` is not directly usable.

`weak_ptr` is convertible back to `shared_ptr` for use if the object is still in existence.

C++11 `move semantics` (`&&`) improve performance of ownership transfer.
Safe array passing: `std::array` / `boost::array`

C++ array problems

```cpp
class Image {
    // Unsafe; size unknown
    uint8_t[] getLUT();
        void setLUT(uint8_t[] & lut);
};
```

- C++ arrays “decay” to “bare” pointers
- Pointers have no associated size information
Safe array passing: `std::array / boost::array`

`std::array`

```cpp
// Safe; size defined
typedef std::array<uint8_t, 256> LUT;
const LUT& getLUT() const;
void setLUT(const LUT& lut);
```

- `std::array` is an array-like object
- `std::array` size defined in the template
- `std::array` can be passed like any object
- Bounds checking with `.at()`
- Unchecked access with `[]`
Containers storing different types

Types with a common base

```cpp
std::vector<std::shared_ptr<Base> > v;
v.push_back(std::make_shared<Derived>());
```

- Store any type derived from `Base`

Java containers can be problematic

- Java can store root `Object` in containers
- Java can pass and return root `Object` in methods.
- This isn't possible in C++: there is no root object.
- An alternative approach is needed.
Containers storing different types

**Arbitrary types: boost::any**

```cpp
boost::any value = Anything;
std::vector<boost::any> v;
v.push_back(Anything);
```

- Assign and store any type
- Type erasure (similar to Java generics)
- Use for containers of arbitrary types
- Flexible, but need to cast to each type used to extract
- Code won't be able to handle all possible types meaningfully
Containers storing different types

Fixed set of types: `boost::variant`

```cpp
typedef boost::variant<int, std::string> variants;
std::vector<variants> v;
v.push_back(43);
v.push_back("ATTO 647N")
```

- Store a set of discriminated types
- “External polymorphism” via visitors
- Used to store original metadata
- Used to store nD pixel data of different pixel types
```cpp
namespace {  
    MetadataMap map;
}
void test() {  
    MetadataMap flat_map (map.flatten());
}

MetadataMap MetadataMap::flatten() const {
    MetadataMap newmap;

    for (MetadataMap::const_iterator i = oldmap.begin();
         i != oldmap.end(); ++i) {
        MetadataMapFlattenVisitor v(newmap, i->first);
        boost::apply_visitor(v, i->second);
    }

    return newmap;
}
```

- Visitor created for each key in the original map
- Visitor applied once using the value in the original map
- The value type (i->second) selects the correct visitor template
Basic language differences
Reference handling and memory management
Containers

Arrays
Storing different types
variant examples

MetadataMap boost::variant visitor pattern (1)

```
// Flatten MetadataMap vector values
struct MetadataMapFlattenVisitor : public
    boost::static_visitor<> {
  MetadataMap& map; // Map of flattened elements
  const MetadataMap::key_type& key; // Current key

  MetadataMapFlattenVisitor
    (MetadataMap& map,
     const MetadataMap::key_type& key):
    map(map), key(key) {}

  // Output a scalar value of arbitrary type.
  template <typename T>
  void operator() (const T& v) const {
    map.set(key, v);
  }

  // Function operator expanded for every variant scalar type
```
MetadataMap boost::variant visitor pattern (2)

// Output a vector value of arbitrary type.
template <typename T>
void operator() (const std::vector<T>& c) const {
    typename std::vector<T>::size_type idx = 1;
    for (typename std::vector<T>::const_iterator i = c.begin();
        i != c.end(); ++i, ++idx) {
        std::ostringstream os;
        os << key << " #" << idx;
        map.set(os.str(), *i);
    }
};

- Function operator expanded for every variant vector type
- Vectors are split into individual scalar values in the new map
void test() {
    VariantPixelBuffer a, b;
    if (a == b) {
        // Buffers are the same.
    }
}

bool VariantPixelBuffer::operator == (const VariantPixelBuffer& rhs) const {
    return boost::apply_visitor(PBCompareVisitor(),
                                 buffer, rhs.buffer);
}

- VariantPixelBuffer contains any supported pixel type.
- Pixel comparison only performed for compatible types.
VariantPixelBuffer comparison visitor

```cpp
struct PBCompareVisitor : public boost::static_visitor<bool> {
    template <typename T, typename U>
    bool operator() (const T& /* lhs */, const U& /* rhs */) const {
        return false;
    }

    template <typename T>
    bool operator() (const T& lhs, const T& rhs) const {
        return lhs && rhs && (*lhs == *rhs);
    }
};
```

- Comparisons of different types always `false`
Acknowledgements

- OME Team, Dundee
  - Jason Swedlow
  - Jean-Marie Burel
  - Mark Carroll
  - Andrew Patterson
  - …and the rest of the team

- Micron, Oxford
  - Douglas Russell

- Glencoe Software
  - Melissa Linkert
  - Josh Moore