KDAB

Introduction to the Boost C++ Libraries

for KDE developers

Volker Krause
volker@kdab.com

What is Boost?

- Comprehensive set of platform-independent C++ libs
- http://www.boost.org
- Free Software
- About 100 modules, 80% header-only
- Staging ground for the next C++ standard library
- Pushes C++ to its limits

Whirlwind Tour through Boost

KDAB

- Containers
- Data Structures
- Iterators
- Algorithms
- Function Objects
- Higher Order Programming
- Generic Programming
- Template Metaprogramming
- Preprocessor Metaprogramming
- Text Processing
- Parser Generation
- Concurrency
- Math and Statistics
- Image Processing
- Platform Abstraction
- Python Binding Generation
- Unit testing
- ...

Boost vs. Qt/KDE

- No GUI components
- Some overlap with Qt/KDE (~ 20 modules):
  - Platform abstraction
  - Signals
  - Date/Time, RegExp, Serialization
  - Smart Pointers (since Qt 4.6)
- Ease of use vs. flexibility
- Documentation
- STL-style naming

#1 usage in KDE: Smart Pointers

KDAB

- Feature equivalent smart pointers in Qt since 4.6
- Widely known concept and widely used in KDE already
  - Shared Pointer
  - Scoped Pointer
  - Weak Pointer
- More powerful than you might think...

Custom Deleter

```
QMutex *mutex = ...;
{
  boost::shared_ptr<QMutex> mutex_releaser( mutex,
    std::mem_fun(&QMutex::unlock) );
  mutex->lock();
  ...
}
```

#2 usage in KDE: boost::bind

- Many STL and Qt algorithms require a function pointer or function object as argument
- Cumbersome when done manually:

```
static void laterDeleter( QObject *obj ) {
  obj->deleteLater();
}
// possible lots of other code
QList<QObject*> l = ...;
std::for_each( l.begin(), l.end(), laterDeleter );
```

Define function objects in-place

In-place member call:

```
std::for_each( l.begin(), l.end(),
  boost::bind(&QObject::deleteLater, _1) );
```

In-place member call with arguments:

```
QObject *parent = ...;
std::for_each( l.begin(), l.end(),
  boost::bind(&QObject::setParent, _1, parent ) );
```

Lame, show me something useful!

Sort by arbitrary properties of an object:

```
QList<QObject*> l = ...;

qSort( l.begin(), l.end(),
  boost::bind(&QObject::objectName, _1) <
  boost::bind(&QObject::objectName, _2) );
```

So, how does this work exactly?

- boost::bind() returns a function object
- Number of arguments depend on number of used placeholders (_X)
- boost::bind(&f, a, _2, b, _1)(x, y) → f(a, y, b, x)
- Works for global and member functions (ie. no std::mem_fun needed, first argument is the object)
- There are overloaded operators for these function objects: !, ==, !=, <, <=, >, >=, &&, ||
- Cascading is possible as well

Slightly more complex: boost::graph

- We work a lot with tree/graph structures, mostly without realizing that though
- We rarely use graph algorithms though
- Boost has a comprehensive graph library
- Requires slightly more work to use, due to lack of explicit graph data structures and/or standardized interfaces (like we have for lists)
- Example: Find the most specific type from a set of mime-types

Create a graph...

```
QVector<PluginInfo> plugins;
const PluginInfo& findBestMatch( KmimeType::Ptr mimeType ) {
  boost::adjacency_list<> graph( matchingIndexes.size() );
  for ( int i = 0, end = plugins.size() ; i != end ; ++i ) {
    for ( int j = 0; j != end; ++j ) {
      if ( i != j && mimeType->is( plugins[j].mimeType() ) )
        boost::add_edge( j, i, graph );
    }
  }
  ...
```

... and sort it

```
  ...
  QVector<int> order;
  order.reserve( plugins.size() );
  try {
    boost::topological_sort( graph,
      std::back_inserter( order ) );
  } catch ( boost::not_a_dag &e ) {
    kWarning() << "Mimetype tree is not a DAG!";
  }
  return plugins[order.first()];
}
```

Compatibility with Qt

- Qt containers are STL compatible
- #undef QT_NO_STL
- Container-like structures such as QString and QByteArray have STL support for reading but not for writing:

```
QList<QByteArray> list = ...;

QByteArray result = boost::join( list, “, “ );
```

Conclusions

- Do not reinvent the wheel! *)
- Steep learning curve, but it will pay off nevertheless

But also keep in mind:

- Prefer equivalent Qt classes
- Limit use in public API (no BC guarantees)

*) Unless when having a strong NIH policy

Thanks for listening!

KDAB

Questions?

Special thanks to my colleagues Marc Mutz, Stephen Kelly and Kevin Ottens for voluntarily or involuntarily providing content for this talk :)