C++11 tour
What will change for library designers?

Maxime van Noppen
maxime.van.noppen@gmail.com
https://github.com/mvannop/cpp11_tour

24 July 2012

Table of contents

Outline

The big picture: what could change?

Adopting C++11 should hopefully help us to:

- Improve maintainability: less overhead, less bloat

The big picture: what could change?

Adopting C++11 should hopefully help us to:

- Improve maintainability: less overhead, less bloat
- Improve performances: new ways to manipulate objects

The big picture: what could change?

Adopting C++11 should hopefully help us to:

- Improve maintainability: less overhead, less bloat
- Improve performances: new ways to manipulate objects
- Create new designs enabling new features: new OO and generic language features

The big picture: what could change?

Adopting C++11 should hopefully help us to:

- Improve maintainability: less overhead, less bloat
- Improve performances: new ways to manipulate objects
- Create new designs enabling new features: new OO and generic language features

At the end of the day:

- Development costs should decrease for the library developer **and** the library user

The big picture: what could change?

Adopting C++11 should hopefully help us to:

- Improve maintainability: less overhead, less bloat
- Improve performances: new ways to manipulate objects
- Create new designs enabling new features: new OO and generic language features

At the end of the day:

- Development costs should decrease for the library developer **and** the library user
- Product value should increase

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...
- ...but recent compilers (g++ 4.7, llvm 3.1, MSVC 11) are moving fast...

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...
- ...but recent compilers (g++ 4.7, llvm 3.1, MSVC 11) are moving fast...
- ...and this is great compared to C++98/C++03 compilers lag.

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...
- ...but recent compilers (g++ 4.7, llvm 3.1, MSVC 11) are moving fast...
- ...and this is great compared to C++98/C++03 compilers lag.
- `https://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport`

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...
- ...but recent compilers (g++ 4.7, llvm 3.1, MSVC 11) are moving fast...
- ...and this is great compared to C++98/C++03 compilers lag.
- `https://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport`
- Using C++11 feels like exploring a new language

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...
- ...but recent compilers (g++ 4.7, llvm 3.1, MSVC 11) are moving fast...
- ...and this is great compared to C++98/C++03 compilers lag.
- `https://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport`

- Using C++11 feels like exploring a new language
- No "best practices" set in stone yet

The big picture: Work in progress

- Fully-compliant compilers not expected before 2013...
- ...but recent compilers (g++ 4.7, llvm 3.1, MSVC 11) are moving fast...
- ...and this is great compared to C++98/C++03 compilers lag.
- `https://wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport`

- Using C++11 feels like exploring a new language
- No "best practices" set in stone yet
- C++1x (C++17?) is already in the tubes

Outline

Problem

```
template <typename T>
Matrix<T> operator+(const Matrix<T>& rhs,
                    const Matrix<T>& lhs)
{
  Matrix<T> res;
  /* ... */
  return res;
}
```

- Copying a temporary might be very expensive
- No easy solution
- We have to hope that "return value optimisation" (RVO) kicks in
- What happens with complex formulas mixing a lot of operators?

- Avoid deep-copying temporaries

- Avoid deep-copying temporaries
- More reliable than ”return value optimisation” (RVO)

- Avoid deep-copying temporaries
- More reliable than "return value optimisation" (RVO)
- Eases writing of overloaded operators on complex types

```
template <typename T>
Matrix<T> operator+(const Matrix<T>& rhs,
                    const Matrix<T>& lhs)
{
  Matrix<T> res;
  /* ... */
  return std::move(res);
}
```

- Avoid deep-copying temporaries
- More reliable than "return value optimisation" (RVO)
- Eases writing of overloaded operators on complex types

```
template <typename T>
Matrix<T> operator+(const Matrix<T>& rhs,
                    const Matrix<T>& lhs)
{
  Matrix<T> res;
  /* ... */
  return std::move(res);
}
```

- Simply recompiling code with a move-enabled STL may increase performances

- Create a move constructor

```
template <typename T> class Matrix
{
  public:
    Matrix(int n, int m) : data(new T[n * m]) { }
    ~Matrix() { delete data; }
    Matrix(Matrix&& tmp)
    {
      data = tmp.data;
      tmp.data = 0;
    }

    T* data;
};
  Matrix<double> m(3, 3);
  // m.data == 0x9973008
  Matrix<double> n = std::move(m);
  // m.data == 0
  // n.data == 0x9973008
```

Outline

1. Big picture
   - What could change?
   - Work in progress
2. Rvalue references
3. Variadic templates
4. Rvalue references + variadic templates
5. Conclusion

Problem

What is a **generic** function?

Problem

What is a **generic** function?

- In C++03: generic on the **type** of its arguments

```
template <typename T> void foo(T t) { }
```

Problem

What is a **generic** function?

- In C++03: generic on the **type** of its arguments

```
template <typename T> void foo(T t) { }
```

- But not generic on the **number** of its arguments

Problem

What is a **generic** function?

- In C++03: generic on the **type** of its arguments

```
template <typename T> void foo(T t) { }
```

- But not generic on the **number** of its arguments
- What about the good old printf-style functions?

Problem

What is a **generic** function?

- In C++03: generic on the **type** of its arguments

```
template <typename T> void foo(T t) { }
```

- But not generic on the **number** of its arguments
- What about the good old printf-style functions?

Challenge: write a 'min' function which takes an arbitrary number of arguments of any type and returns the smallest in C++03.

Problem

What is a **generic** function?

- In C++03: generic on the **type** of its arguments

```
template <typename T> void foo(T t) { }
```

- But not generic on the **number** of its arguments
- What about the good old printf-style functions?

Challenge: write a 'min' function which takes an arbitrary number of arguments of any type and returns the smallest in C++03.

To support up to `N` arguments there needs to be `N` overloads...

Introducing variadic templates

- Combination of the old '. . . ' (think of printf) and classic templates

Introducing variadic templates

- Combination of the old '. . . ' (think of printf) and classic templates
- What about our challenge?

Introducing variadic templates

- Combination of the old '...' (think of printf) and classic templates
- What about our challenge?

```
template <typename T>
const T& min(const T& a, const T& b)
{
  return a < b ? a : b;
}

template <typename T, typename... Args>
const T& min(const T& a, const T& b, const Args&... args)
{
  return min(a, min(b, args...));
}
```

- Very efficient thanks to inlining

A typesafe printf!

```
template <typename T>
void print(const T& t)
{
  std::cout << t;
}
template <typename T, typename... Ts>
void print(const T& t, const Ts&... tail)
{
  print(t);
  print(tail...);
}

print("There are ", 3, " arguments!", std::endl);
```

Outline

Problem

Writing a generic "make_shared" function which **constructs** the object.

Either:

- Create **a lot** of overloads...

Problem

Writing a generic "make_shared" function which **constructs** the object.
Either:

- Create **a lot** of overloads...
- Add constraints on the type, typically being default constructible

Problem

Writing a generic "make_shared" function which **constructs** the object.
Either:

- Create **a lot** of overloads...
- Add constraints on the type, typically being default constructible
- ...don't construct the object but take a pointer to an already constructed object...

```
boost::shared_ptr<std::string> sptr =
  boost::make_shared_ptr(new std::string("..."));
```

Perfect forwarding to the rescue!

- Rvalue references + variadic templates = perfect forwarding

```
template <typename T, typename... Args>
std::shared_ptr<T> make_shared(Args&&... args)
{
  std::shared_ptr<T> p(new T(std::forward<Args>(args)...));
  return p;
}
std::shared_ptr<std::string> sptr =
  make_shared<std::string>("...");
```

Perfect forwarding to the rescue!

- Rvalue references + variadic templates = perfect forwarding

```
template <typename T, typename... Args>
std::shared_ptr<T> make_shared(Args&&... args)
{
  std::shared_ptr<T> p(new T(std::forward<Args>(args)...));
  return p;
}
std::shared_ptr<std::string> sptr =
  make_shared<std::string>("...");
```

- Used in the STL to create emplace_* methods

```
int main()
{
  std::vector<std::string> strings;
  strings.emplace_back("This is a test!");
  strings.emplace_back(42, 'a');
}
```

Outline

Conclusion

"The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever" — Bjarne Stroustrup

`http://www2.research.att.com/~bs/C++0xFAQ.html`

Conclusion

"The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever" — Bjarne Stroustrup

`http://www2.research.att.com/~bs/C++0xFAQ.html`

As library designers:

- More powerful and safe interfaces
- Better performances with simpler designs
- And this with a better maintainability

Conclusion

"The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever" — Bjarne Stroustrup

`http://www2.research.att.com/~bs/C++0xFAQ.html`

As library users:

- More powerful and safe interfaces
- Better performances with simpler designs
- And this with a better maintainability

Conclusion

"The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever" — Bjarne Stroustrup

`http://www2.research.att.com/~bs/C++0xFAQ.html`

There is much more!

- We only scratched the surface of C++11
- Code bloat should decrease while features should increase
- We will see new paradigms emerge from the new core features

Bibliography

- C++ Now! (former BoostCon) `http://cppnow.org`
  - **Leor Zolman**: A Whirlwind Overview of C++11
  - **Alisdair Meredith**: Lessons Learned Developing the C++11 Standard Library
  - **Howard Hinnant**: What's new with C++11 containers?
  - **Scott Schurr**: C++11: New Tools for Class and Library Authors
  - `http://cppnow.org/schedule-table/`
- **Bjarne Stroustrup**: `http://www2.research.att.com/~bs/`
- **C++Next**: `http://cpp-next.com/`

Questions?

Mixins

```
struct HasPosition
{
  double path_cost(int dest) const { /*...*/ }
  int position;
};
struct HasShape { /* ... */ };

template <typename... Mixins>
class Object : public Mixins...
{
};

int main()
{
  Object<HasPosition> waypoint;
  Object<HasPosition, HasShape> building;
}
```

Type inference

```
template <template <class...> class Cont,
          typename Fun, typename... Args>
auto map(const Cont<Args...>& c, Fun f)
  -> Cont<decltype(f(*c.begin()))>
{
  typedef decltype(f(*c.begin())) map_type;
  Cont<map_type> out;
  for (const auto& e : c)
    out.emplace_back(f(e));
  return std::move(out);
}

int main()
{
  std::vector<int> v = { 42, 51 };
  auto s = map(v, [](int n) { return std::to_string(n); });
}
```