Diet for Your Templates

Tips for Reducing Code Bloat

Sebastian Redl
C++Now 2016

About Me

- Work for TEOCO, Vienna office
- C++ programmer for 18 years
- Maintainer of Boost.PropertyTree

Background

- Created IoC++ for work project
- Dependency Injection library with high runtime flexibility
- Supports separate compilation
- Dedicated presentation tomorrow

Background

- Uses lots of templates
- Linker died from out-of-memory errors
- Had to reduce the size of the object files

About IoC++

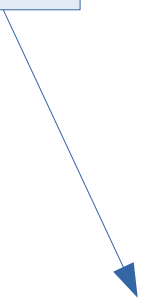
```
struct A { virtual ~A() {} };
struct B : A { int i; B(int i) : i(i) {} };
struct C {
    std::shared_ptr<A> a;
    C(std::shared_ptr<A> a) : a(a) {}
};
struct D {
    std::shared_ptr<C> c;
    D(std::shared_ptr<C> c) : c(c) {}
};
```

About IoC++

```
Container container;
container.configure(

// reflection information
    type<B>(constructorDependencies(
        valueArgument<int>("i"))).provides<A>(),
    type<C>(constructorDependencies(
        objectArgument<A>("a"))),
    type<D>(constructorDependencies(
        objectArgument<C>("c"))),

// object graph configuration
    object("").ofType("B").map("i").to(1),
    object("2").ofType("B").map("i").to(2)
    );

// retrieval
auto c = container.resolve<C>(); // shared_ptr<C>
```

About IoC++

```
Container container;
container.configure(

// reflection information
    type<B>(constructorDependencies(
        valueArgument<int>("i"))).provides<A>(),
    type<C>(constructorDependencies(
        objectArgument<A>("a"))),
    type<D>(constructorDependencies(
        objectArgument<C>("c"))),

// object graph configuration
    object("").ofType("B").map("i").to(1),
    object("2").ofType("B").map("i").to(2)
    );
```



This generates a lot of code!
We want to reduce that.

Setup

- Measure size of several files:
  - libutil.lib, compiled IoC++ sources and others
  - DecoratorTest.obj, most complex unit test
  - VeloxLib.lib, main internal user
- Sizes include debug information
- Very inexact tests: lots of unrelated code
- Size optimization was intermingled with speed optimization, which sometimes undid wins

Techniques

- Avoid complex templates
- Factor out type-independent code
  - Pay runtime price if it's worth it
- Explicitly instantiate templates
- Algorithmic improvements still trump everythir

Avoid Complex Templates

- std::function generates a lot of code
- A simple virtual function can do the job

Avoid Complex Templates

- Lifestyle Manager: implements lifestyles
  - Singleton: one object for everyone
  - Transient: new object on every request
  - Session: same object for duration of “session”
- User Extension Point

Avoid Complex Templates

```
class LifestyleManager
{
public:
    virtual ~LifestyleManager() {}

    using Creator = std::function<std::shared_ptr<void>()>;

    virtual std::shared_ptr<void>
        get(const Creator& creator) = 0;
};
```

VeloxLib.lib size:
873,629 kB

Avoid Complex Templates

```
class LifestyleManager
{
public:
    virtual ~LifestyleManager() {}

    struct Creator {
        virtual std::shared_ptr<void> create() = 0;
    };

    virtual std::shared_ptr<void> get(Creator& creator) = 0;
};
```

VeloxLib.lib size:
857,380 kB

Avoid Complex Templates

```
template <typename T>
struct LifestyledProvider<T>::CallCreate
{
public:
    CallCreate(const LifestyledProvider<T>* provider,
               ResolutionContext* context,
               bool* wasUsed)
        : provider(provider), context(context), wasUsed(wasUsed)
    { *wasUsed = false; }

    std::shared_ptr<void> operator ()()
    {
        *wasUsed = true;
        try {
            auto result = provider->createObject(context);
            return result;
        } catch (...) { // trace information is added here
            throw;
        }
    }

private:
    const LifestyledProvider<T>* provider;
    ResolutionContext* context;
    bool* wasUsed;
};
```

Avoid Complex Templates

```
template <typename T>
struct LifestyledProvider<T>::CallCreate :
    public lifestyles::LifestyleManager::Creator
{
public:
    CallCreate(const LifestyledProvider<T>* provider, ResolutionContext& context)
        : provider(provider), context(context), wasUsed(false)
    {}

    bool used() const { return wasUsed; }

    std::shared_ptr<void> create() override
    {
        wasUsed = true;
        try {
            auto result = provider->createObject(context);
            return result;
        } catch (...) { // trace information is added here
            throw;
        }
    }

private:
    const LifestyledProvider<T>* provider;
    ResolutionContext& context;
    bool wasUsed;
};
```

Avoid Complex Templates

- Side effect: less malloc traffic
  - CallCreate allocated on stack in new version
  - Too big for std::function SBO
- Another instance of this optimization saved 40MB in VeloxLib.lib size

Avoid Complex Templates

- Base class list can contain pattern expansion

```
template <typename ActualType, typename... ProvidedTypes>
class MultiTypeProvider :
    public InheritAll2<ConvertingProvider, ActualType,
                       ProvidedTypes...>
```

- InheritAll2 uses recursion
- This is simpler and saves 4MB

```
template <typename ActualType, typename... ProvidedTypes>
class MultiTypeProvider :
    public ConvertingProvider<ProvidedTypes, ActualType>...
```

De-Templatize All The Things

- Template code often doesn't depend on parameters
- Extract this code to separate non-template functions
- May require some refactoring

De-Templatize All The Things

```
template <typename T>
T resolveArgument(ResolutionContext& context,
    const configuration::ValueArgument<T>& argument,
    const InternalObjectConfiguration& config)
{
    try {
        auto result = resolveValueArgument(context, argument, config);
        context.getTracer().resolveArgumentEnd(true);
        return result;
    } catch (exceptions::ContainerException& e) {
        e.traceResolve(argument.getDependencyName());
        context.getTracer().resolveArgumentEnd(false);
        throw;
    } catch (...) {
        context.getTracer().resolveArgumentEnd(false);
        throw;
    }
}
```

De-Templatize All The Things

```
template <typename T>
boost::any getValueSeed(
    const configuration::ValueArgument<T>& argument,
    const InternalObjectConfiguration& config)
{
    auto value = config.getValueFor(argument.getDependencyName());
    if (value.empty())
        value = argument.getDefaultValue();
    return value;
}

template <typename T>
T resolveValueArgument(ResolutionContext& context,
    const configuration::ValueArgument<T>& argument,
    const InternalObjectConfiguration& config)
{
    auto value = getValueSeed(argument, config);
    context.getRegistry().convertValue(
        context, argument.getDependencyName(), typeid(T), value);
    return boost::any_cast<T>(value);
}
```

De-Templatize All The Things

```
template <typename T>
class ValueArgument : public detail::DependencyArgument
{
public:
    using argument_type = T;

    explicit ValueArgument(std::string dependencyName)
        : DependencyArgument(std::move(dependencyName))
    {}

    ValueArgument& defaultsTo(T defaultValue)
    {
        this->defaultValue = defaultValue;
        return *this;
    }

    const boost::any& getDefaultValue() const { return defaultValue; }

private:
    boost::any defaultValue;
};
```

De-Templatize All The Things

```
class ValueArgumentBase : public DependencyArgument
{
public:
    ValueArgumentBase(std::string&& dependencyName)
        : DependencyArgument(std::move(dependencyName))
    {}

    const boost::any& getDefaultValue() const { return defaultValue; }

protected:
    boost::any defaultValue;
};
```

De-Templatize All The Things

```
template <typename T>
class ValueArgument : public detail::ValueArgumentBase
{
public:
    using argument_type = T;

    explicit ValueArgument(std::string&& dependencyName)
        : ValueArgumentBase(std::move(dependencyName))
    {}

    ValueArgument& defaultsTo(T defaultValue)
    {
        this->defaultValue = defaultValue;
        return *this;
    }
};
```

De-Templatize All The Things

```
boost::any getValueSeed(const ValueArgumentBase& argument,
                        const InternalObjectConfiguration& config)
{
    auto value = config.getValueFor(argument.getDependencyName());
    if (value.empty())
        value = argument.getDefaultValue();
    return value;
}

boost::any resolveValueArgumentCore(
    ResolutionContext& context,
    const std::type_info& type,
    const ValueArgumentBase& argument,
    const InternalObjectConfiguration& config)
{
    auto value = getValueSeed(argument, config);
    context.getRegistry().convertValue(context,
        argument.getDependencyName(), type, value);
    return value;
}
```

De-Templatize All The Things

```
boost::any resolveValueArgument(
    ResolutionContext& context,
    const std::type_info& type,
    const ValueArgumentBase& argument,
    const InternalObjectConfiguration& config)
{
    try {
        auto value = resolveValueArgumentCore(
            context, type, argument, config);
        context.getTracer().resolveArgumentEnd(true);
        return value;
    } catch (exceptions::ContainerException& e) {
        e.traceResolve(argument.getDependencyName());
        context.getTracer().resolveArgumentEnd(false);
        throw;
    } catch (...) {
        context.getTracer().resolveArgumentEnd(false);
        throw;
    }
}
```

De-Templatize All The Things

```
template <typename T>
T resolveArgument(ResolutionContext& context,
    const configuration::ValueArgument<T>& argument,
    const InternalObjectConfiguration& config)
{
    return boost::any_cast<T>(
        resolveValueArgument(context, typeid(T), argument, config));
}
```

De-Templatize All The Things

- Two more similar functions refactored
- Cumulative change
  - Saved 14MB in VeloxLib.lib
  - Added 400kB in libutil.lib

De-Templatize All The Things

- You can transform compile time polymorphism to runtime polymorphism
- Usually means an additional virtual call
- Can be worth it for the size savings
  - Instruction cache behavior might even make it faster

De-Templatize All The Things

```
template <typename T>
std::shared_ptr<T> Registry::decorate(
    const std::shared_ptr<T>& object, ResolutionContext& context)
{
    auto decorators = findDecorators(typeid(T));
    if (decorators.first == decorators.second)
        return object;

    if (decorators.first->second.active)
        return object;
    ScopedSetter<bool> activateDecorator(decorators.first->second.active, true);

    auto wrapped = object;
    for (auto it = decorators.first; it != decorators.second; ++it) {
        auto& decorator = it->second;

        Provider* decoratorProvider =
            decorator.findExistingDecoratorProvider(wrapped);
        if (!decoratorProvider) {
            auto newProvider = createProviderForConfig(
                decorator.getDefinition(), context);
            decorator.saveExistingDecoratorProvider(object, newProvider);
            decoratorProvider = newProvider.get();
        }
        ScopedDecorationContext sdc(context, wrapped);
        wrapped = objectFromProvider<T>(context, *decoratorProvider);
    }
    return wrapped;
}
```

De-Templatize All The Things

```
std::shared_ptr<void> Registry::decorateCore(
    ResolutionContext& context, std::shared_ptr<void> wrapped,
    const std::type_info& type, const FromProvider& fromProvider)
{
    auto decorators = findDecorators(type);
    if (decorators.first == decorators.second)
        return wrapped;

    if (decorators.first->second.active)
        return wrapped;
    ScopedSetter<bool> activateDecorator(decorators.first->second.active, true);

    for (auto it = decorators.first; it != decorators.second; ++it) {
        auto& decorator = it->second;

        Provider* decoratorProvider =
            decorator.findExistingDecoratorProvider(wrapped);
        if (!decoratorProvider) {
            auto newProvider = createProviderForConfig(
                decorator.getDefinition(), context);
            decorator.saveExistingDecoratorProvider(wrapped, newProvider);
            decoratorProvider = newProvider.get();
        }
        ScopedDecorationContext sdc(context, wrapped);
        wrapped = fromProvider.get(context, *decoratorProvider);
    }
    return wrapped;
}
```

De-Templatize All The Things

```
struct FromProvider {
    virtual std::shared_ptr<void> get(
        ResolutionContext& context, Provider& provider) const = 0;
};

template <typename T>
std::shared_ptr<T> Registry::decorate(
    const std::shared_ptr<T>& object, ResolutionContext& context)
{
    class TypedFromProvider : public FromProvider
    {
    public:
        explicit TypedFromProvider(Registry& registry)
          : registry(registry) {}
        std::shared_ptr<void> get(ResolutionContext& context,
                                  Provider& provider) const override {
            return registry.objectFromProvider<T>(context, provider);
        }

    private:
        Registry& registry;
    };
    return std::static_pointer_cast<T>(decorateCore(
        context, object, typeid(T), TypedFromProvider(*this)));
}
```

De-Templatize All The Things

- Modest gains (1.2MB in VeloxLib.lib)
- One virtual call per decorator
  - The common case is no decorators at all
- Another instance of this refactoring saved 4M
  - Costs 3 virtual calls and 1 dynamic cross-cast
  - The latter could have been traded for larger memory footprint

Explicitly Instantiate Templates

- C++11 added extern template declarations
- Prevent the compiler from instantiating a template
- Explicit instantiation puts the generated code i a central location
- Useful when you use some template component with fixed arguments

Explicitly Instantiate Templates

- IoC++ builds conversion paths
  - string (name of object)
  - object configuration
  - instantiated object
- Edges of graph are functions doing conversion
  - name lookup gets from string to configuration
  - configuration instantiation yields actual object
- Graph component is used with fixed arguments

Explicitly Instantiate Templates

```
template <typename Vertex, typename Edge>
class PathFinder;

template <typename... ExtraConverterArgs>
class TypeConversion
{
    using ConversionFunction =
        std::function<void (boost::any&, ConversionStatus&,
                            ExtraConverterArgs...)>;
    using Converter = std::pair<ConversionFunction, std::string>;
    using Finder = PathFinder<std::type_index, Converter>;
    Finder finder;
};

using ValueConverter = TypeConversion<ResolutionContext&>;
using ObjectConverter = TypeConversion<ResolutionContext&,
                                       const ConversionOptions&,
                                       const UpdateConfig&>;
```

Explicitly Instantiate Templates

```
extern template TypeConversion<ResolutionContext&>;
extern template TypeConversion<ResolutionContext&, const UpdateConfig&>;

extern template PathFinder<std::type_index,
    TypeConversion<ResolutionContext&>::Converter>;
extern template PathFinder<std::type_index,
    TypeConversion<ResolutionContext&, const UpdateConfig&>::Converter>;
```

Explicitly Instantiate Templates

```
template TypeConversion<ResolutionContext&>;
template TypeConversion<ResolutionContext&, const UpdateConfig&>;

template PathFinder<std::type_index,
    TypeConversion<ResolutionContext&>::Converter>;
template PathFinder<std::type_index,
    TypeConversion<ResolutionContext&, const UpdateConfig&>::Converter>;
```

Explicitly Instantiate Templates

- Initially, this didn't save anything!
- All the member functions of the class template were defined inline
  - Compiler still instantiated them in every source
- Had to outline the member function definitions

Explicitly Instantiate Templates

```
template <typename... ExtraConverterArgs>
class TypeConversion
{
    ConversionResult convert(boost::any& value,
        std::type_index targetType, ExtraConverterArgs... args) const
    {
        const auto& sourceType = value.type();
        auto paths = finder.find(targetType, sourceType);
        ConversionStatus status(sourceType, targetType);
        for (auto& path : paths) {
            status.startPath(describePath(path));
            tryPath(status, path, value, args...);
            if (status)
                break;
        }
        return status.toResult();
    }
};
```

Explicitly Instantiate Templates

```
template <typename... ExtraConverterArgs>
class TypeConversion
{
    ConversionResult convert(boost::any& value,
        std::type_index targetType, ExtraConverterArgs... args) const;
};

template <typename... ExtraConverterArgs>
ConversionResult TypeConversion<ExtraConverterArgs...>::convert(
    boost::any& value, std::type_index targetType,
    ExtraConverterArgs... args) const
{
    const auto& sourceType = value.type();
    auto paths = finder.find(targetType, sourceType);
    ConversionStatus status(sourceType, targetType);
    for (auto& path : paths) {
        status.startPath(describePath(path));
        tryPath(status, path, value, args...);
        if (status)
            break;
    }
    return status.toResult();
}
```

Explicitly Instantiate Templates

- VeloxLib.lib shrunk 23MB
- libutil.lib grew 1.5MB

Algorithm Trumps Everything

- At this point, I had achieved ~20% size saving
- Just kept up with size increase from new functionality
- Realized I had overlooked a fundamental improvement

Algorithm Trumps Everything

```
Container container;
container.configure(

// reflection information
    type<B>(constructorDependencies(
        valueArgument<int>("i"))).provides<A>(),
    type<C>(constructorDependencies(
        valueArgument<float>("f"))).provides<A>(),
    type<D>(constructorDependencies(
        valueArgument<std::string>("s"))).provides<A>(),
    type<E>(constructorDependencies(
        objectArgument<A>("a"))),
    );

// retrieval
auto e = container.resolve<E>();
```

Algorithm Trumps Everything

- Generating all conversion code for every registered type (A, B, C, D, E)
- Only A and E are ever queried
- Rearranged internals so that code is only generated for A and E
- VeloxLib.lib dropped from 843MB to 490MB

Lessons Learned

- Optimizing templates is similar to optimizing runtime
- Measure every step – I threw away a few things
- Look for algorithmic improvements first
- Separate argument-dependent and argument-independent code
  - Even if you don't extract it to a source file, it gives the linker a chance to fold different versions together

Lessons Learned

- Explicitly instantiating templates only helps when member functions are outlined
  - But then it can help a lot
- std::function is nice for user interfaces, but expensive in terms of code size

Questions?