Pure C++ options classes.

Supporting constructors with optional, typed named actual arguments, in class hierarchies.

A GUI widget such as a button or a listbox typically has a great many options, more than can be reasonably specified as individual positional arguments. With a suitable script language that supports named actual arguments it's easy to specify values for the relevant options when the widget is created, using defaults for the other options, and this works in practice because usually only a few options need non-default values. But how can this be done in the spirit of C++ with strict, static type checking?

The named parameter idiom solution.

The **named parameter idiom**, I'll refer to it as **NPI**, discussed in the C++ FAQ[1] provides a simple solution for any given single class `T`. Essentially a `TOptions` class is defined with a setter member function for each supported `T` option, where each such setter function returns a `TOptions&` reference to the `TOptions` object. The setter calls can then be chained, and the resulting option values object passed to a `T` constructor.

A `T` instance can then be constructed like …

```
T aTObject( TOptions()
      .text( "blah" )
      .width( 120 )
      );
```

The `TOptions` class may be named to provide more readable source code, suggesting the higher level purpose of the most common use case, like the FAQ's (Marshall Cline's) example where `T` is a class `File` and `TOptions` is a class named `OpenFile`, …

```
File f = OpenFile( "foo.txt" )
            .readonly()
            .createIfNotExist()
            .appendWhenWriting()
            .blockSize( 1024 )
            .unbuffered()
            .exclusiveAccess();
```

But note that in this example `OpenFile` does not open the file, it only supplies the parameters – the name `OpenFile` just indicates the overall effect of this use case.

To support other use cases a better or alternative name might be `OpenFileParams`.

Alternatively a `TOptions` class can be named systematically as e.g. a nested class `T::Params`, which in the FAQ's example would be `File::Params`. Or there can be many option value classes for a given class `T`, e.g. one per `T` constructor, with each options class name indicating which constructor one invokes, e.g. `File::OpenParams`. And so on; many variations are possible, the above only introduces the main idea.

Why NPI is not enough: the covariant setters problem.

When `T` is a class in a class hierarchy chances are that its immediate base class has an options set, and that `T`'s option set should be an extension. As a concrete example, when modeling the Windows API's GUI widgets you may have an `AbstractButton` class with options `hTextAlign`, `vTextAlign` and `buttonPlacement`, and among others a concrete derived class `CheckBox` with additional options `is3State` and `isAuto`. And then, especially for the base class initializations, it would be nice if `CheckBox::Params` was derived from `AbstractButton::Params`; then e.g. the `CheckBox` constructor can just pass its `Params` argument up to `AbstractButton`.

But with direct use of the named parameter idiom that doesn't work.

For since `AbstractButton::Params::hTextAlign` returns `AbstractButton::Params&` you can't chain a call to e.g. `CheckBox::Params::isAuto` (this is the minimal example involving only one derived class, and with only one option defined per class):

covariance_problem.cpp

```
// This code intentionally fails to compile at line 42.

struct AbstractButton
{
    struct Params
    {
        int hTextAlignValue;
        Params() : hTextAlignValue( 0 ) {}

        int hTextAlign() const
        { return hTextAlignValue; }

        Params& hTextAlign( int v )
        { hTextAlignValue = v; return *this; }
    };

    explicit AbstractButton( Params const& = Params() )
    {}
};

struct CheckBox: AbstractButton
{
    struct Params: AbstractButton::Params
    {
        bool isAutoValue;
        Params(): isAutoValue( true ) {}

        bool isAuto() const
        { return isAutoValue; }

        Params& isAuto( bool v )
        { isAutoValue = v; return *this; }
    };

    explicit CheckBox( Params const& params = Params() )
        : AbstractButton( params )
    {}
};

int main()
{
    CheckBox    widget( CheckBox::Params()
                    .hTextAlign( 1 )    // OK
                    .isAuto( false )    // Oops! ERROR! Unknown!
                    );
}
◁
```

Ah, but no problem, just let the client (calling) code always specify the base class options *last*, yes? Ufortunately, no, that doesn't work either. For then, in the example above the actual argument will be of static type `AbstractButton::Params`, and the `CheckBox` constructor doesn't accept that: it needs a `CheckBox::Params` argument.

A manual setters overloading solution.

One cure is to manually overload all the inherited setters in the derived options class, changing the return types to "follow" the class, that is, with **covariant** return types:

solution_using_manual_overloading.cpp

```
    ⋮
struct CheckBox: AbstractButton
{
    struct Params: AbstractButton::Params
    {
        bool isAutoValue;
        Params(): isAutoValue( true ) {}

        int hTextAlign() const
        { return AbstractButton::Params::hTextAlign(); }

        Params& hTextAlign( int v )
        {
            return static_cast<Params&>(
                AbstractButton::Params::hTextAlign( v )
                );
        }

        bool isAuto() const         { return isAutoValue; }
        Params& isAuto( bool v )    { isAutoValue = v; return *this; }
    };

    explicit CheckBox( Params const& = Params() )
        : AbstractButton( params )
    {}
};
    ⋮
◁
```

The reason that also the `hTextAlign` getter, not only the setter, is overloaded here, is that *any* `hTextAlign` overload hides or "shadows" the base class implementations.

This hiding rule is so perplexing to C++ novices that it has its own FAQ entry[2], but without it the setter overloading couldn't work since one can't overload solely on return type (which brings in a terminology issue: is "overload" the right word?).

Why manual setters overloading can be impractical.

Manually implemented setters work, but the source code size overhead of introducing a single new option in a derived class can be quite large when the base class has many options. And when a new option is introduced in a base class, or an option is renamed or removed, then all the derived classes' options classes have to be manually updated. If a base class has n options, and has a chain of d derivations down to some class, each of which introduces 1 new option, then the number of setter implementations for that chain is $(d+1)\cdot m + (d^2+d)/2$. If every class in that chain

of $n = d+1$ classes defines m options, then the number of setter implementations is $m \cdot (n^2+n)/2$. And even though we're talking very small numbers here this means that the manual setters overloading approach can often be wholly impractical.

A Boost Parameter Library based solution.

Usually when confronted with a very general problem that one can imagine that there must be a portable solution for, a good place to start looking for a solution is the Boost library. And indeed Boost has a named parameters sublibrary, the **Boost Parameter Library**[3] by David Abrahams and Daniel Wallin, which I'll refer to as the **BPAL**, that at least partially fits the bill. The BPAL is, naturally, based on templating.

With the usual NPI options are represented as named data members, like …

```
struct Options
{
    int     a, b, c;
    Options( int aValue = 1, int bValue = 2, int cValue = 3 )
        : a( aValue ), b( bValue ), c( cValue )
    {}
};

void foo()
{
    Options o;
    cout << o.a << o.b << o.c << endl;
}
```

… but one key idea of the BPAL approach is that instead of named data members one represents each logical option data member as a *base class subobject*, like …

```
struct A { int value; A( int v = 1 ): value( v ) {} };
struct B { int value; B( int v = 2 ): value( v ) {} };
struct C { int value; C( int v = 3 ): value( v ) {} };
struct Options: A, B, C {};

void foo()
{
    Options o;
    cout << o.A::value << o.B::value << o.C::value << endl;
}
```

… where the two versions of `foo` have the exact same effect, producing "123".

In BPAL the `Options` is named `ArgumentPack`, and instead of directly containing option values it contains logical references to the actual arguments of a function call.

With each logical data member represented as base class subobject the logical data member is referenced by specifying its **tag type**, like `A`, `B` and `C` above. This supports sorting the arguments of a function where the arguments are of templated types and specified in an arbitrary order by the caller, because each argument value, or for efficiency a pointer or reference to each argument value, can be copied to the right place in an `Options` object simply by using that argument's tag type as identification. And this argument sorting capability in turn supports writing or generating *templated wrappers* of a function so that named actual arguments can be specified in any order.

BPAL provides a number of boilerplate code generation macros for such templated forwarding wrappers. However, since C++98 does not support constructor forwarding the concept of wrapping and forwarding does not play well with C++98 constructors. Hence, in the example code below, for the client code's constructor calls I rely exclusively on the BPAL's support for constructing `ArgumentPack` instances via the comma operator, which introduces an extra set of parentheses like `(( … ))`:

solution_using_boost_named_parameters.cpp

```
#include <iostream>
#include <boost/parameter.hpp>

typedef boost::parameter::aux::empty_arg_list   EmptyArgumentPack;

namespace abstractButton {
    BOOST_PARAMETER_NAME( hTextAlign )
}  // namespace abstractButton

struct AbstractButton
{
    int     hTextAlign;

    struct Params
    {
        int     hTextAlign;

        template< class ArgumentPack >
        Params( ArgumentPack const& args )
            : hTextAlign( args[abstractButton::_hTextAlign | 0] )
        {}
    };

    explicit AbstractButton(
        Params const& params = Params( EmptyArgumentPack() )
        )
        : hTextAlign( params.hTextAlign )
    {}
};

namespace checkBox {
    using namespace abstractButton;
    namespace tag { using namespace abstractButton::tag; }
    BOOST_PARAMETER_NAME( isAuto )
}  // namespace checkBox

struct CheckBox: AbstractButton
{
    bool    isAuto;

    struct Params: AbstractButton::Params
    {
        bool    isAuto;

        template< class ArgumentPack >
        Params( ArgumentPack const& args )
            : AbstractButton::Params( args )
            , isAuto( args[checkBox::_isAuto | true] )
        {}
    };

    explicit CheckBox(
        Params const& params = Params( EmptyArgumentPack() )
        )
        : AbstractButton( params )
        , isAuto( params.isAuto )
    {}
};
```

```
void show( CheckBox const& cb )
{
    using namespace std;
    cout
        << boolalpha
        << "align = " << cb.hTextAlign << ", "
        << "isAuto = " << cb.isAuto << ".\n";
}

int main()
{
    using namespace checkBox;

    CheckBox    widget1;
    show( widget1 );        // 0, true (the default values)

    CheckBox    widget2((
                    _hTextAlign = 1
                    ));
    show( widget2 );        // 1, true

    CheckBox    widget3((
                    _hTextAlign = 1,
                    _isAuto = false
                    ));
    show( widget3 );        // 1, false
}
```

Here the nested `Params` classes serve to (1) centralize the specification of default values; (2) allow custom defaults; and (3) support handling options separately, without having an instance of e.g `CheckBox`. I chose this solution because apparently for constructors the default value specification must done where each value is extracted from a BPAL `ArgumentPack`, using an `|` operator, instead of more like a declaration. For example, without the `Params` as unpacking + default value helper `AbstractButton` would need a templated constructor and a separate default constructor, with redundant specification of the default values in each constructor.

It might seem as if this forces a redundant declaration of each named parameter, e.g. `AbstractButton::hTextAlign` plus `AbstractButton::Params::hTextAlign` of the same type. But in real code there might not be an `AbstractButton::hTextAlign` member. Instead that might for example be a state of some underlying API level widget.

Why the BPAL based solution can be undesirable.

Like the manual overloading of setters the BPAL based approch works, but also like that previous approach it's not ideal; there are some problems:

- *Avalanche-like cryptic diagnostics*: results from the heavy templatization.
- *Large library dependency*: using the BPAL forces use of the Boost library (sometimes Boost can't be used, e.g. due to company or project policy).
- *Special client code support, "gruff"*: at least with the solution above, which is about the best that I could do with BPAL, the client code has to introduce extra supporting namespaces; it has to deal directly with BPAL things such as

`ArgumentPack`, some of it, like `empty_arg_list`, apparently undocumented; and it has to use unconventional BPAL-specific notation such as the `|` operator.

- *"Ingenious" code*: the pattern can be difficult to understand for maintainers.
- *Too lenient type checking*: e.g. the `CheckBox` constructor will accept an `ArgumentPack` with options unrelated to `CheckBox` provided that it doesn't contain anything incompatible with the `CheckBox::Params` expectations.

The ordinary forwarding-wrapper-functions use of BPAL does not suffer from the type checking leniency problem: the wrappers perform checking, and BPAL supplies macros and class templates to construct the necessary information for that checking.

However, BPAL's only macro for generating constructor wrappers, called `BOOST_PARAMETER_CONSTRUCTOR`, assumes & requires a base class constructor as the implementation to forward to. In the example above that would preclude having `CheckBox` derived from `AbstractButton`. Functionality for checking whether a general `ArgumentPack` produced in a call is a subset of the supported arguments, plus the ability to extend declarations of what arguments are supported, could remedy this problem, but alas, it seems that such functionality is not yet present in BPAL.

An automated setter overloading solution.

The problems of Boost dependency, special support "gruff", "ingenious" code, and type checking leniency, can be solved by automating the generation of `Options` setter overloads, where the code generation macros directly generate the desired result – an `Options` class – and nothing more that the client code needs relate to:

solution_using_automated_overloading.cpp

```
#include <iostream>
#include <progrock/cppx/collections/options.h>

struct AbstractButton
{
    int     hTextAlign;

    CPPX_DEFINE_1OPTIONCLASS( Params, CPPX_OPTIONS_NO_BASE
        , hTextAlign    , int       , 0
        )

    explicit AbstractButton( Params const& params = Params() )
        : hTextAlign( params.hTextAlign() )
    {}
};

struct CheckBox: AbstractButton
{
    bool    isAuto;

    CPPX_DEFINE_1OPTIONCLASS( Params, AbstractButton::Params
        , isAuto        , bool      , true
        )

    explicit CheckBox( Params const& params = Params() )
        : AbstractButton( params )
        , isAuto( params.isAuto() )
    {}
};
```

```
void show( CheckBox const& cb )
{
    using namespace std;
    cout
        << boolalpha
        << "align = " << cb.hTextAlign << ", "
        << "isAuto = " << cb.isAuto << ".\n";
}

int main()
{
    CheckBox    widget1;
    show( widget1 );        // 0, true (the default values)

    CheckBox    widget2( CheckBox::Params()
                    .hTextAlign( 1 )
                    );
    show( widget2 );        // 1, true

    CheckBox    widget3( CheckBox::Params()
                    .hTextAlign( 1 )
                    .isAuto( false )
                    );
    show( widget3 );        // 1, false
}
◁
```

Here `progrock` is a namespace & directory that I use for my code in general.

The **[options.h]** header, a pure header file module with nothing that needs separate compilation, supplies `CPPX_DEFINE_`*n*`OPTIONCLASS` macros with *n* from 1 to 9 inclusive. These macros support arbitrarily deep levels of options class derivation. The most that I've actually used has however been just four levels of options class derivation.

These macros are all that's needed in order to use this functionality, it's very simple to use: just invoke the appropriate macro for your number of options.

Why the setter generation macros can be annoying.

As with the manual setters overloading approach the number of generated setter implementations is in a sense quadratic in the depth of class derivation, which might sound alarming – uh oh, quadratic number of setters…

However, the depth of class derivation is usually very small, like maximum 4 levels of derivation, and so, while "quadratic" does have some negative vibes this isn't a practical problem; it's the compiler that deals with it, and it deals well with it.

More practically, with the `CPPX_DEFINE_`*n*`OPTIONCLASS` macros, when you add or remove an option you have to update *n* accordingly. In contrast to the "quadratic" this might not *seem* to be a problem, it's just a few keystrokes. But a wrong *n* generally causes an avalanche of cryptic error messages, since like the BPAL the [options.h] header relies on templatization. And even though the first error message will tell you something about wrong number of arguments you might not realize that it's a wrong *n*, and thus waste some time, perhaps quite a lot of time. Plus, it's rather annoying to have to count things when the computer should excel at that task.

A solution with effectively variadic C++98 macros.

Happily the **Boost Preprocessor Library**, some truly amazing and very impressive code written by Paul Mensonides, can make the C++ preprocessor, even the severely limited C++98 preprocessor, sing, dance, compose sonnets, and in general almost anything, it's almost unbelievable!, including that it supports a limited form of variadic macros for C++98 – so that options can be added or removed freely.

However, since this re-introduces a dependency on Boost I chose to put the Boost'ed macro in a separate header **[options_boosted.h]**, which in turn includes [options.h]. If one already uses Boost then [options_boosted.h] provides convenience at little or no extra cost. If one doesn't already use Boost then the [options.h] *n*-macros do work.

Using [options_boosted.h], instead of writing …

```
// "manual n" macro from [options.h]
CPPX_DEFINE_3OPTIONCLASS( Params, CPPX_OPTIONS_NO_BASE
    , hTextAlign        , int       , 0
    , vTextAlign        , int       , 0
    , buttonPlacement   , int       , 0
    )
```

… you use a slightly different syntax and write …

```
// "automatic n" macro from [options_boosted.h]
CPPX_DEFINE_OPTIONCLASS( Params, CPPX_OPTIONS_NO_BASE,
    ( hTextAlign,       int,        0   )
    ( vTextAlign,       int,        0   )
    ( buttonPlacement,  int,        0   )
    )
```

… with no *n* in sight, and no need to count anything, generating the same code.

`CPPX_DEFINE_OPTIONCLASS` supports from 1 to 11 options, inclusive. The limit of 11 is due to a limit of 12 directly specified types in the non-Boost typelist used internally. There is no way to specify "no options" via this macro, because a Boost Preprocessor Library sequence – the list of horizontal parentheses above – can't be empty.

How it works: tricky recursive macros and templates.

The Boost Preprocessor Library macros include functionality for iterating over simple sequences like `(a)(b)(c)` at preprocessing time, with a specified macro as a kind of callback – and yes, I'm talking about C++98 macros that iterate, iteration at preprocessing time ☺. However, the Boost Preprocessor Library iteration macros and other sequence handling macros fail when the sequence is a *sequence of tuples* like `(a,int,1)(b,int,2)(c,int,3)`, as in `CPPX_DEFINE_OPTIONCLASS`' third argument. Trying my hand at doing this myself I came to appreciate Paul Mensonides' sheer genius in creating that kind of preprocessor magic in the first place – i.e., I failed.

But I'd seen that BPAL does this, to the extreme… So to create [options_boosted.h] I sort of reverse-engineered some BPAL support macros and vigorously chopped off irrelevant stuff, and thus wrt. BPAL ended up using only apparently undocumented

functionality from [boost/parameter/aux_/preprocessor/for_each.hpp]. I used that to convert `(a,int,1)(b,int,2)(c,int,3)` to `((a,int,1))((b,int,2))((c,int,3))`, and then the ordinary Boost Preprocessor Library macros could do the rest.

This additional internal Boost magic, a macro called `BOOST_PARAMETER_FOR_EACH_R` that can iterate over a sequence of tuples, was written by Daniel Wallin.

Now about the rest, the strictly not-Boost-dependent code.

A `CPPX_DEFINE_`*n*`OPTIONCLASS` invocation expands to code that uses template based code generation.

The setter overloads are brought in by inheritance from classes templated on the desired setter result type. Each such class defines setter and getter overloads for one option. The particular option is identified by a tag type, as in the BPAL approach.

And this means that also each option's logical data member (a base class sub-object) is necessarily brought in by inheritance, as in the BPAL approach.

However, the setter overloads need to be generated anew for each options class *T*, to have *T*`&` result type, while each option logical data member should only be brought in once. So they need to be treated differently. Each logical option data member *name* is therefore generated as an ordinary class *T*`_cppx_OptionValue_`*name*, while the generic definition of that option's setter and getter is generated as class template `cppx_OptionSetter_`, partially specialized on the option's logical data member class.

For example, the earlier example program's macro invocation down in `CheckBox`, …

```
CPPX_DEFINE_1OPTIONCLASS( Params, AbstractButton::Params
    , isAuto        , bool       , true
    )
```

… generates, among other things, …

```
class Params_cppx_OptionValue_isAuto
    : public ::progrock::cppx::options::Value_<
        bool,
        struct cppx_UniqueIdTypeFor_Params_isAuto
        >
{
    typedef ::progrock::cppx::options::Value_<
        bool,
        struct cppx_UniqueIdTypeFor_Params_isAuto
        > Base;
public:
    Params_cppx_OptionValue_isAuto()
        : Base( true )
    {}

    Params_cppx_OptionValue_isAuto( bool const& v )
        : Base( v )
    {}
};

template< class OptionValue, class SetterResult, class Base >
class cppx_OptionSetter_;
```

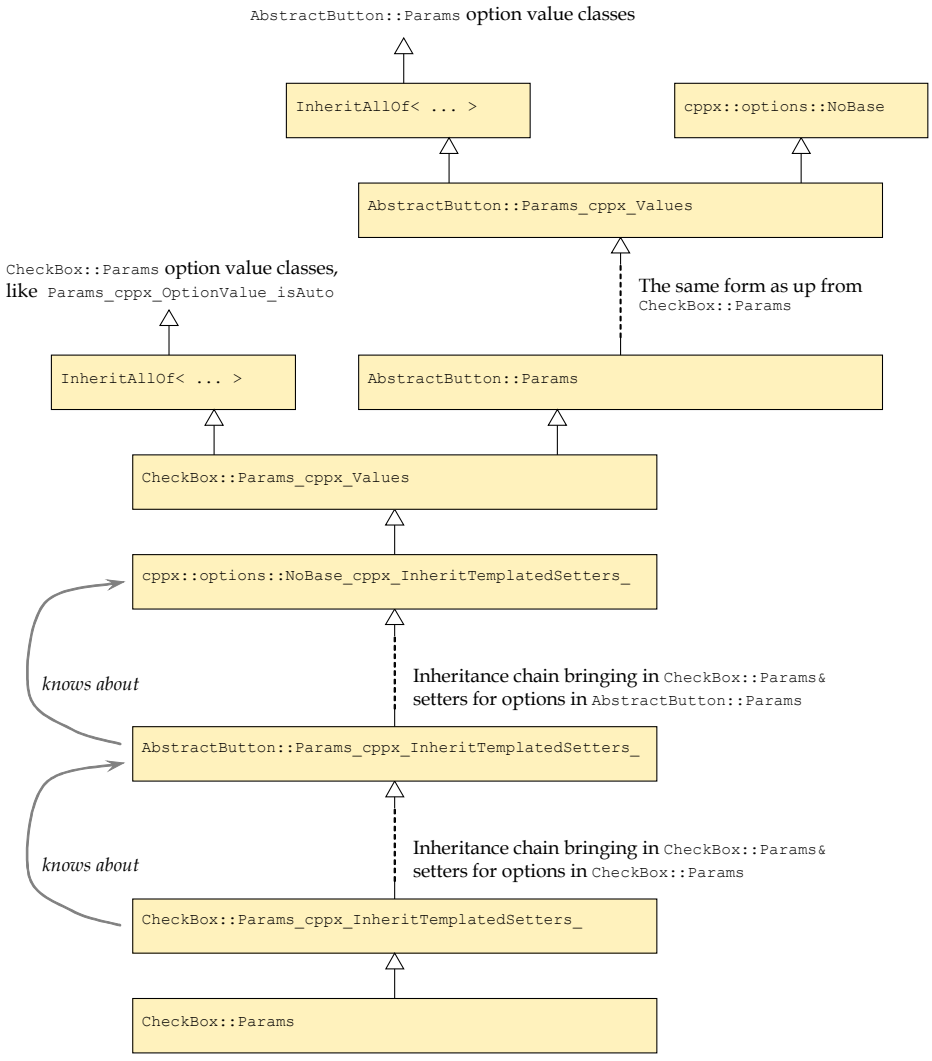
```
template< class SetterResult, class Base >
class cppx_OptionSetter_< Params_cppx_OptionValue_isAuto, SetterResult, Base >
    : public Base
{
public:
    bool const& isAuto() const
    {
        return Params_cppx_OptionValue_isAuto::value();
    }

    SetterResult& isAuto( bool const& value )
    {
        Params_cppx_OptionValue_isAuto::setValue( value );
        return ::progrock::cppx::downcast< SetterResult >( *this );
    }

    SetterResult& set_isAuto( bool const& value )
    {
        return isAuto( value );
    }
};
```

The general inheritance tree assembling these two pieces per option is shown in figure 1 (unfortunately this is an instance of “a picture’s worth a thousand words”):

Figure 1: the general inheritance tree.



For each options class *T* the *T*-specific inheritance – of *T*'s base class, appropriate setter overloads for *T*'s own options, and *T*'s own logical option data members – is accomplished by a generated class template `T_cppx_InheritTemplatedSetters_`:

```
template< class SetterResult, class TopBase >
class Params_cppx_InheritTemplatedSetters_
    : public ::progrock::cppx::options::InheritTemplatedSetters_<
        Params_cppx_OptionTypes,
        SetterResult,
        AbstractButton::Params_cppx_InheritTemplatedSetters_<
            SetterResult, TopBase
            >,
        cppx_OptionSetter_
        >
{};
```

`T_cppx_InheritTemplatedSetters_` delegates the main inheritance work to `::progrock::cppx::options::InheritTemplatedSetters_` (static code, not generated by the macro), asking that general `InheritTemplatedSetters_` class template to bring in *T*'s base class *U*'s `U_cppx_InheritTemplatedSetters_`, which in turn will delegate its work back to `InheritTemplatedSetters_` asking it to bring in *U*'s base class *V*, and so on, in a back and forth indirect recursion.

The base case for that particular recursion is …

```
namespace progrock{ namespace cppx{ namespace options {

    class NoBase {};

    template< class SetterResult, class TopBase >
    class NoBase_cppx_InheritTemplatedSetters_: public TopBase {};

} } }  // namespace progrock::cppx::options
```

… but as you can see, while it stops recursing up the base class chain to bring in setter overloads, hurray, it still itself inherits from `TopBase`. What's that? Well, it's the option values for the bottom options class *T* and all its options class bases.

Deferring consideration of `TopBase`, the general `InheritTemplatedSetters_`:

```
namespace progrock{ namespace cppx{ namespace options {

    template<
        class OptionTypes,
        class SetterResult,
        class TopBase,
        template< class, class, class > class OptionSetter_
        >
    class InheritTemplatedSetters_;

    template<
        class SetterResult,
        class TopBase,
        template< class, class, class > class OptionSetter_
        >
    class InheritTemplatedSetters_<
        cppx::EmptyTypelist, SetterResult, TopBase, OptionSetter_
        >
        : public TopBase
    {};
```

```
    template<
        class OptionTypes,
        class SetterResult,
        class TopBase,
        template< class, class, class > class OptionSetter_
        >
    class InheritTemplatedSetters_
        : public OptionSetter_<
            typename cppx::HeadOf< OptionTypes >::T,
            SetterResult,
            InheritTemplatedSetters_<
                typename cppx::TailOf< OptionTypes >::T,
                SetterResult,
                TopBase,
                OptionSetter_
                >
            >
    {};

} } }  // namespace progrock::cppx::options
```

At this point you see some glimpses of the non-Boost typelist facility, the `HeadOf` and `TailOf` class templates from **[typelist.h]**. It's very limited but sufficient for purposes like this. If you're unfamiliar with typelists you may just study that header, or learn about them from e.g. Andrei Alexandrescu's *Modern C++ Design*, which is the book that taught me about them – apart from TCP^{3}L it's the single must-have C++ book.

Here `OptionSetter_` is a template as template parameter. Depending on the `InheritTemplatedSetters_` specialization it can be different templates; each options class can have its own with partial specializations for each option. For example, the inheritance above can be of `CheckBox::cppx_OptionSetter_< ..._isAuto, ... >`, as shown earlier, where the chain to inherit further from is passed to the `OptionSetter_`.

Going down to the more concrete, `InheritTemplatedSetters_` is instantiated from each options class' *T*`_cppx_InheritTemplatedSetters_` class template, as shown, …

```
template< class SetterResult, class TopBase >
class Params_cppx_InheritTemplatedSetters_
    : public ::progrock::cppx::options::InheritTemplatedSetters_<
        Params_cppx_OptionTypes,
        SetterResult,
        AbstractButton::Params_cppx_InheritTemplatedSetters_< SetterResult, TopBase >,
        cppx_OptionSetter_
        >
{};
```

… which in turn is instantiated from the concrete options class *T*, like …

```
class Params
    : public Params_cppx_InheritTemplatedSetters_<
        Params,
        Params_cppx_Values
        >
{};
```

Here the `Params_cppx_Values` is macro-generated class that assembles the options (via inheritance delegated to the typelist facility's `InheritAllOf`), …

```
class Params cppx Values
    : public AbstractButton::Params
    , public ::progrock::cppx::InheritAllOf<
        Params_cppx_OptionTypes
        >
{};
```

… where `Params_cppx_OptionTypes` is macro-generated typelist that specifies the options to assemble, …

```
typedef ::progrock::cppx::Typelist<
    Params_cppx_OptionValue_isAuto
    >::T Params_cppx_OptionTypes;
```

And that's about all, sans some details.

Since I myself find this code difficult to grok it is perhaps not all clear to you!

It's a bit intricate, but that's TMP (Template Metaprogramming). ☺

Conclusion.

The code supplied with this article has been tested in Windows XP with the MinGW g++ 3.4.5, MSVC 7.1 and MSVC 9.0 C++ compilers, using Boost library versions 1.35 and 1.42 for testing of [options_boosted.h].

I've not included license comments but the Boost license can be assumed, which essentially means that you can use it freely even in commercial applications. Some earlier versions of the code were posted to the [comp.lang.c++] Usenet group.

For the automated setters generation approach – at least for my take on it here – the generated code is complex, and the supporting fixed code is complex. And much of the fixed code is necessarily in macros, which is not maintainance friendly. And so it's probably not very useful as a general pattern for creating similar things.

Happily the result, like the [options.h] macros, can be very simple to *use*, as shown, providing clear and simple client code without any visible implementation details, enabling you to define class hierarchies with constructors that have optional, strictly typed named actual arguments – for example for GUI widget wrappers. Or, for that matter, just ordinary functions with optional arguments. Also, the [options_boosted.h] header may inspire you to create similar effectively variadic C++98 macros (although you may of course opt for C99/C++0x variadic macros).

Since it's so easy to use there's IMO no need to generate options classes via scripts, complicating and slowing down the build process. Unless there are special requirements it can all be done in pure C++. Enjoy.

Alf P. Steinbach
Oslo, Norway, March 2010

Acknowledgments.

I would especially like to thank James Kanze for brainstorming and critique in a discussion of these ideas in the [comp.lang.c++] Usenet group, some long time ago.

– EOT –

[1] At the time of writing <url: http://www.parashift.com/c++-faq-lite/ctors.html#faq-10.18>.
[2] At the time of writing <url: http://www.parashift.com/c++-faq-lite/strange-inheritance.html#faq-23.9>.
[3] At the time of writing <url: http://www.boost.org/doc/libs/1_36_0/libs/parameter/doc/html/index.html>.