Chimeric Pointer

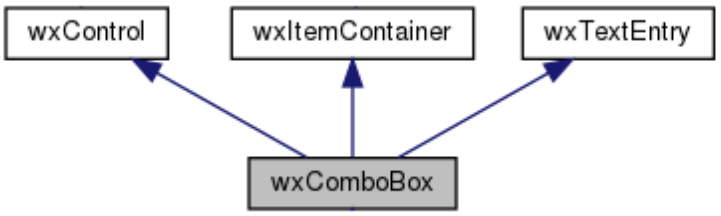
2022-11-27

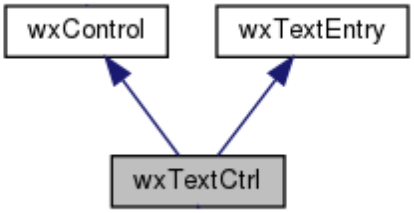
(Draft No. 3 by T. P. K. Healy)

I was writing code for a graphical user interface program on a desktop PC using the ***wxWidgets*** framework. I had designed a dialog box that had several widgets, in particular text boxes and combo boxes. I wanted to write a function that could manipulate either a combo box or a text box, as follows:

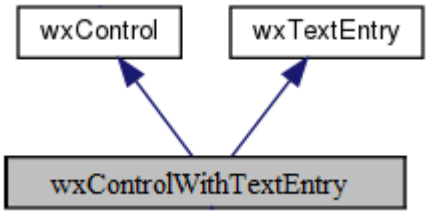
```
void Red(wxControl *const p)
{
    p->SetBackgroundColour( *wxRED );
    p->SetValue("pending");
    p->Refresh();
}
```

This function didn't compile however, because '*SetValue*' is not a member function of the class **wxControl**. So I took a look at the ***wxWidgets*** documentation and I found the class hierarchy diagrams:





Text boxes and combo boxes both inherit the '*SetValue*' method from **wxTextEntry**. It would have been nice if there were an intermediate class called **wxControlWithTextEntry** as follows:



because then both **wxComboBox** and **wxTextCtrl** could inherit from **wxControlWithTextEntry**, and so then my function '*Red*' above could take a pointer to a **wxControlWithTextEntry**.

Not wanting to write a template function, I had to re-write my function as follows:

```
void Red(wxControl *const pC, wxTextEntry *const pT)
{
    pC->SetBackgroundColour( *wxRED );
    pT->SetValue("pending");
    pC->Refresh();
}
```

and then invoke it as follows:

```
wxTextCtrl *const p1 = new wxTextCtrl;
Red(p1,p1);

wxComboBox *const p2 = new wxComboBox;
Red(p2,p2);
```

This is when I came up with the idea of a '*chimeric pointer*'. A chimeric pointer would work as follows:

```
void Red( chimeric_pointer<wxControl,wxTextEntry> p )
{
    p->SetBackgroundColour( *wxRED );
    p->SetValue("pending");
    p->Refresh();
}
```

The way that this chimeric pointer would work is as follows:

(1) When defining a chimeric pointer, you specify all of the base classes you need, for example:

```
chimeric_pointer<wxControl,wxTextEntry> p;
```

(*Note that the order in which you specify the Base classes is important for the look-up*)

(2) When assigning to a chimeric pointer, the expression on the right-hand side must be a pointer to a class which can convert implicitly to a pointer to all of the base classes specified in the first point above, otherwise the compiler will terminate compilation and issue a diagnostic message.

(3) When you apply the '->' operator to a chimeric pointer and then try to access a member object or a member function, the compiler tries to find the member object/function in all of the base classes specified in the first point above.

So for example, in the above code snippet where I have:

```
p->SetBackgroundColour( *wxRED );
```

The compiler searches for a member named '*SetBackgroundColour*' in **wxControl**, and it successfully finds such a method and invokes it. For the next example, let's take the next line:

```
p->SetValue("pending");
```

The compiler searches for a member called '*SetValue*' in **wxControl**, and it fails to find it. So next it searches for a member called '*SetValue*' in **wxTextEntry**, and it finds it and invokes it.

If the compiler cannot find the member inside any of the base classes, then compilation is terminated and the compiler must issue a diagnostic.

Alternatively we could write the function '*Red*' as a template function as follows:

```
template<class T>
requires (   std::is_convertible_v<T*, wxControl  *>
          && std::is_convertible_v<T*, wxTextEntry*>)
void Red(T *const p)
{
    p->SetBackgroundColour( "red" );
    p->SetValue("pending");
    p->Refresh();
}
```

There are four drawbacks to having a template function:

(*Drawback No. 1*) The size of the machine code increases as there will be an instantiation of '*Red*' for each class (e.g. **wxTextCtrl**, **wxComboBox**, **wxSomeOtherWidget**)

(*Drawback No. 2*) If the body of '*Red*' contains a definition of a static-duration object, there will be a copy of the object for each of the instantiations. If the program is multi-threaded, there will also be a mutex and lock-management code to prevent double-construction of the static-duration object.

(*Drawback No. 3*) If the function body of '*Red*' contains a definition of a static-duration **std::mutex** to protect a global object, then there will be more than one mutex (i.e there will be one mutex for each instantiation of '*Red*').

(*Drawback No. 4*) We don't have just one function pointer that can be invoked on a pointer to any class that derives from both **wxControl** and **wxTextEntry**.

I will go into further detail about *Drawback No. 4*. Let's say we have an array of function pointers that we want to invoke on an object, something like:

```
void (*func_ptrs[3u])(chimeric_ptr<wxControl,wxTextEntry>) = { Red, Green, Blue };
```

And let's say we make use of this array in an event handler as follows:

```
void Dialog_Main::OnClick_Stop(wxCommandEvent&)
{
    for ( wxControl *const p_control : p_controls )
    {
        for ( auto const f : func_ptrs ) f(p_control);
    }
}
```

This is only possible if there is just one function in memory that can deal with all classes which derive from both **wxControl** and **wxTextEntry**.

With regard to the look-up of members, here is a complex case:

```
struct AirBreather { int close; };

struct WaterBreather { float (*close)(char); };

struct Frog : virtual AirBreather, virtual WaterBreather {};

int main(void)
{
    Frog my_frog;

    chimeric_pointer<AirBreather,WaterBreather> p = &my_frog;

    auto x = p->close;
}
```

In the above code snippet, '**x**' is a variable of type `int`.

However if we re-order the base classes as follows:

```
    chimeric_pointer<WaterBreather,AirBreather> p = &my_frog;
```

then '**x**' is now a pointer to a function which takes a `char` and returns a `float`. The compiler searches for the member in the base classes in the order in which they're written from left to right. When performing a lookup, the compiler stops searching as soon as it finds a match. The following code will fail to compile because 'close' is an `int`:

```
chimeric_pointer<AirBreather,WaterBreather> p = &my_frog;
p->close('k');   // COMPILER ERROR - 'close' is an int
```

On 2022-11-27 on the C++ Standard Proposals Mailing List, Marian Darius provided sample code for an implementation of **chimeric_ptr** which would be very similar to what I am proposing. Darius's code works as follows:

```
    chimeric_ptr<WaterBreather,AirBreather> p = &my_frog;

    p.as<AirBreather>()->close = 5;

    p.as<WaterBreather>()->close('n');
```

I have made additions to Darius's code to make it work with complex virtual inheritance, which you can see up on the ***GodBolt*** website here: https://godbolt.org/z/csMKTjjsh
On the next page you can see the entire code copy-pasted from ***GodBolt***.

```
// BEGIN Chimeric pointer implementation
#include <type_traits>  // is_convertible_v
#include <tuple>        // tuple
#include <exception>    // exception

// I have chosen to use 'is_convertible_v' instead of "is_base_of_v" for two reasons:
// (1) The latter would accommodate non-public inheritance (we don't want that)
// (2) Currently the C++ programming language only allows Derived class pointers
//     to be converted to Base class pointers. However in the future, maybe the
//     Standard will be changed to allow some other kind of implicit conversion.
//     I want the chimeric pointer to be versatile and future-proof so I'm
//     choosing 'is_convertible_v' over the alternatives of "is_base_of_v" or
//     'derived_from'.

// The next line is the exception that will be thrown if you
// try to de-reference a chimeric_ptr that is a nullptr
class exception_chimeric_nullptr : public std::exception {};

template<class... Bases>
class chimeric_ptr {
protected:

    std::tuple<Bases*...> pointers;

public:

    template<class T>
    requires ((std::is_convertible_v<T*, Bases*> && ...))
    /* implicit */ chimeric_ptr(T *const p)
    {
        // The fold expression on the next line sets
        // each of the pointers in the tuple
        ((std::get<Bases*>(pointers) = p), ...);
    }

    /* implicit */ chimeric_ptr(std::nullptr_t const p)
    {
        // The fold expression on the next line sets
        // each of the pointers in the tuple
        ((std::get<Bases*>(pointers) = nullptr), ...);
    }
```

```
    template<class As>
    requires ((std::is_same_v<Bases, As> || ...))
    As *as(void)
    {
        As *const p = std::get<As*>(pointers);

        if ( nullptr == p ) throw exception_chimeric_nullptr();

        return p;
    }

    bool operator==(std::nullptr_t) const
    {
        return nullptr == std::get<0u>(pointers);
    }
};
// END Chimeric pointer implementation

// Example
struct Control {
    virtual ~Control() = default;

    void Refresh();
    void SetBackgroundColour(const char*);
};

struct TextEntry {
    virtual ~TextEntry() = default;

    void SetValue(const char*);
};

struct TextControl : Control, TextEntry {
    virtual ~TextControl() = default;
};

struct Combo : Control, TextEntry {
    virtual ~Combo() = default;
};
```

```
void Red( chimeric_ptr<Control,TextEntry> p )
{
    if ( nullptr == p ) return;

    p.as<Control>()->SetBackgroundColour( "red" );
    p.as<TextEntry>()->SetValue("pending");
    p.as<Control>()->Refresh();
}

// The following is the alternative (i.e. to have a template function)
template<class T>
requires (   std::is_convertible_v<T*, Control  *>
          && std::is_convertible_v<T*, TextEntry*>)
void Red_Temp(T *const p)
{
    p->SetBackgroundColour( "red" );
    p->SetValue("pending");
    p->Refresh();
}

// Invocations
void UseRed(TextControl* textCtrl, Combo* combo)
{
    Red(textCtrl);
    Red(combo);

    Red_Temp(textCtrl);
    Red_Temp(combo);
}

// The following code is a complex look-up case

struct AirBreather { int close; };

struct WaterBreather { float (*close)(char); };

class Frog : virtual public AirBreather, virtual public WaterBreather {};
```

```
typedef int wxCommandEvent;

class Dialog_Main {
    void OnClick_Stop(wxCommandEvent &event);
};

chimeric_ptr<Control,TextEntry> controls[5u] =
    { nullptr, nullptr, nullptr, nullptr, nullptr};

void (*func_ptrs[3u])(chimeric_ptr<Control,TextEntry>) = { Red, Red, Red };

void Dialog_Main::OnClick_Stop(wxCommandEvent &event)
{
    for ( auto &control : controls )
    {
        for ( auto const &f : func_ptrs )
        {
            f(control);
        }
    }
}

int main(void)
{
    Frog my_frog;

    chimeric_ptr<WaterBreather,AirBreather> p = &my_frog;

    p.as<AirBreather>()->close = 5;
    float some_value = p.as<WaterBreather>()->close('n');

    auto x = p.as<AirBreather>()->close;   // x is of type 'int'

    auto y = p.as<WaterBreather>()->close; // y is of type 'float (*)(char)'
}
```

This implementation of **chimeric_ptr** is quite good but I still think it would be preferable to have compiler support to perform a look-up of the members in the base classes – which would alleviate the need for a member function called '**as**' which must be given a specific base class parameter.

Please respond to this paper on the C++ Standard Proposals Mailing List:

https://lists.isocpp.org/mailman/listinfo.cgi/std-proposals