interface

2023-02-23

(Draft No. 1 by T P K Healy)

This paper is to propose a new feature for the C++ core language to provide an alternative interface for a class without editing the layout of the object. The new '`interface`' keyword can be used to give an alternative interface to a class that's already been defined, for example:

```
interface lockable_bisem : std::binary_semaphore {
    void   lock(void) noexcept(false) { acquire(); }
    void unlock(void) noexcept(false) { release(); }
};

extern std::binary_semaphore g_bisem;

int Func(void)
{
    std::lock_guard<lockable_bisem> myguard(g_bisem);
    // Do more stuff here
}
```

behaves the same as:

```
struct lockable_bisem : std::binary_semaphore {
    void   lock(void) noexcept(false) { acquire(); }
    void unlock(void) noexcept(false) { release(); }
};

extern std::binary_semaphore g_bisem;

int Func(void)
{
    std::lock_guard<lockable_bisem> myguard( *(lockable_bisem*)&g_bisem );
    // Do more stuff here
}
```

Strictly speaking, the second code snippet exhibits undefined behaviour – even though it works fine on every C++ compiler. The reasons why it works fine are discussed overleaf.

Downcasting at compile-time from a **Base** class to a **Derived** class works fine when the following two criteria are met:

1) The **Derived** class does not add any additional member objects
2) The **Derived** class does not define or override any virtual functions

When these two criteria are met, it is always safe to downcast at compile-time from **Base** to **Derived**. In these circumstances, the **Derived** class is free to:

a) Define a non-virtual function with a new name that doesn't exist in the **Base** class
b) Define a non-virtual function to override a non-virtual function of the same name in the **Base** class

An '`interface`' cannot be defined as a template; the following syntax is invalid:

INVALID SYNTAX

```
template<std::ptrdiff_t n>
interface lockable_bisem : public std::counting_semaphore<n> {
    void   lock(void) noexcept(false) { acquire(); }
    void unlock(void) noexcept(false) { release(); }
};
```

However an '`interface`' can provide an alternative interface for a unspecialised template, the following syntax is valid:

VALID SYNTAX

```
interface lockable_countsem : public std::counting_semaphore {
    void   lock(void) noexcept(false) { acquire(); }
    void unlock(void) noexcept(false) { release(); }
};
```

And so then you use '`lockable_countsem`' as though it were a template class:

```
int Func(void)
{
    lock_guard< lockable_countsem<1> > myguard(g_countsem);
    // Do more stuff here
}
```

There is an implicit conversion between a class and any of the '`interface`'s which are based upon it, including an implicit conversion for references and pointers, for example:

```
lockable_bisem *p = &g_bisem;
```

Implicit conversion from `binary_semaphore*` to `lockable_bisem*`

An '`interface`' cannot provide a partial specialisation of a class – instead you must first create a partially-specialised alias, for example:

```
template<class Alloc> using intvector = std::vector<int, Alloc>;
```

And then you can base the '`interface`' upon '`intvector`' as follows:

```
interface popable_intvector : intvector { void pop(void) { this->pop_back(); } };
```

A C++ compiler can perform optimisations when it knows that two or more pointers definitely don't point to the same object (or that two or more references don't refer to the same object). Take the following code snippet shared by Lénárd Szolnoki for example:

```
struct Base {
    int i;
};

struct Derived1 : Base {};
struct Derived2 : Base {};

void foo(Derived1* ptr1, Derived2* ptr2) {
    ptr1->i += ptr2->i;
    ptr1->i += ptr2->i;
    /* compilers currently are allowed to transform this into:
    ptr1->i += 2*ptr2->i;
    */
}
```

When a C++ compiler encounters two pointers where one of the *pointed-to* types is an '`interface`' to the other *pointed-to* type, the compiler must allow for aliasing, for example:

```
template<class Alloc>
void Func( std::vector<int,Alloc> *p, popable_intvector<Alloc> *q )
{
    // Compiler must allow for aliasing between 'p' and 'q'
}
```

A program is ill-formed if two function overloads are identical other than one type being an '`interface`' for another, for example:

```
void Func( std::vector<int, std::allocator<int> >   *);
void Func( popable_intvector< std::allocator<int> > *);
```

> The compiler must terminate compilation and issue a diagnostic to say that an '`interface`' cannot overload for its base class, nor can it overload for another '`interface`' which is based upon the same base class.

In the standard header `<type_traits>`, there shall be a new class: '`is_mutual_interface`' as well as '`is_mutual_interface_v`' which can be used to query whether any type is an '`interface`' for any other type.

More than one ‘interface’ can be based upon the same class, for example:

```
interface popable_string : std::string {
    void pop(void) { pop_back(); }
};

interface reversible_string : std::string {
    void reverse(void)
    {
        for ( unsigned i = 0u; i < (size()/2u); ++i )
        {
            using std::swap;
            swap( (*this)[i], (*this)[size()-1u-i]);
        }
    }
};

interface one_bigger_string : std::string {
    size_t size(void)
    {
        return this->std::string::size() + 1u;
    }
};
```

The compiler is aware of the relationship between these four types, and it must allow for aliasing between the references in the following snippet:

```
#include <type_traits>  // is_mutual_interface

void Func(std::string &a, popable_string &b, reversible_string &c)
{
    // Compiler must allow for aliasing between a, b and c

    // All of the following assertions must succeed
    static_assert( std::is_mutual_interface_v<std::string   ,popable_string   > );
    static_assert( std::is_mutual_interface_v<popable_string,std::string      > );
    static_assert( std::is_mutual_interface_v<std::string   ,reversible_string> );
    static_assert( std::is_mutual_interface_v<popable_string,reversible_string> );
    static_assert( false == std::is_mutual_interface_v<int,double> );

    reversible_string *p1 = &a;  // implicit conversion
    reversible_string *p2 = &b;  // implicit conversion
    popable_string    *p3 = &a;  // implicit conversion
    popable_string    *p4 = &c;  // implicit conversion
    std::string       *p5 = &c;  // implicit conversion
}
```

If an ‘`interface`’ is written to add an additional member object, or if it contains a method defined ‘`virtual`’, or if it contains a method that overrides the base class’s virtual method, the compiler must terminate compilation and issue a diagnostic, as follows:

```
interface how_exception : std::exception {
  char const *how(void) { return what(); }  // This is okay
  int count;                                                  ill-formed (must issue diagnostic)
  virtual bool is_fatal(void) { return false; }               ill-formed (must issue diagnostic)
  char const *what(void) noexcept { return "hello"; } ill-formed (must issue diagnostic)
};
```

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

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

You can view the mailing list archive here:

https://lists.isocpp.org/std-proposals/2023/02/date.php

FIN