noreentry

2023-02-19

(Draft No. 1 by TPK Healy)

I propose a new feature for the C++ core language to prevent the re-entry of functions. The following code snippet:

```
int Func(int const arg) noreentry(return -1)
{
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

behaves the same as:

```
int Func(int const arg)
{
    static std::atomic_flag f = ATOMIC_FLAG_INIT;
    if ( f.test_and_set() ) return -1;
    int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
    f.clear();
    return retval;
}
```

When `noreentry` is applied to a normal function like this, it prevents *all* forms of re-entry – by the same thread and also by other threads. Within the parentheses after `noreentry`, the '`return -1`' means that the body of the function will be skipped (i.e. the attempted re-entry of the function will result in it immediately returning -1). If you wish to allow re-entry by the same thread but not by other threads, the syntax is:

```
int Func(int const arg) noreentry(prevent_others:return -1)
{
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

and it behaves the same as:

```
int Func(int const arg)
{
    static std::recursive_mutex m;
    std::unique_lock<std::recursive_mutex> mylock(m, std::try_to_lock);
    if ( false == mylock.owns_lock() ) return -1;
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

If you want to allow re-entry by other threads but not by the same thread, the syntax is:

```
int Func(int const arg) noreentry(prevent_same:return -1)
{
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

which behaves the same as:

```
int Func(int const arg)
{
    thread_local std::atomic_flag f = ATOMIC_FLAG_INIT;
    if ( f.test_and_set() ) return -1;
    int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
    f.clear();
    return retval;
}
```

In all the examples given up until now, the re-entry has been prevented by simply skipping the body of the function and immediately returning a compile-time constant. Alternatively, re-entry can be prevented by waiting instead of skipping – however this is only possible when preventing re-entry by another thread. The syntax to allow re-entry by the same thread, but to make other threads wait, is:

```
int Func(int const arg) noreentry(prevent_others:wait)
{
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

and it behaves the same as:

```
int Func(int const arg)
{
    static std::recursive_mutex m;
    std::unique_lock<std::recursive_mutex> mylock(m);
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

You can skip for the same thread and wait for other threads with the following syntax:

```
int Func(int const arg) noreentry(prevent_same:return -1,prevent_others:wait)
{
    return (arg >= 0)? (arg + 7) : Func(-arg);
}
```

and it behaves the same as:

```
int Func(int const arg)
{
    static std::recursive_mutex m;
    std::unique_lock<std::recursive_mutex> mylock(m);
    thread_local bool already_entered_by_same_thread = false;
    if ( already_entered_by_same_thread ) return -1;
    already_entered_by_same_thread = true;
    int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
    already_entered_by_same_thread = false;
    return retval;
}
```

All of the examples given up until now have dealt with normal functions, however the re-entry of member functions can also be prevented. The programmer cannot apply the simple keyword `noreentry` to a member function, but instead must apply either `noreentry_this_object` or `noreentry_all_objects`. Examples follow on the next page.

The re-entry of a member function can be prevented **for the same object** with the following syntax:

```
struct MyClass {
    int Func(int const arg) noreentry_this_object(return -1)
    {
        return (arg >= 0)? (arg + 7) : Func(-arg);
    }
};
```

and it behaves the same as:

```
struct MyClass {
    std::atomic_flag f = ATOMIC_FLAG_INIT;
    int Func(int const arg)
    {
        if ( f.test_and_set() ) return -1;
        int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
        f.clear();
        return retval;
    }
};
```

Alternatively the re-entry of a member function can be prevented **for *all* objects** with the following syntax:

```
struct MyClass {
    int Func(int const arg) noreentry_all_objects(return -1)
    {
        return (arg >= 0)? (arg + 7) : Func(-arg);
    }
};
```

and it behaves the same as:

```
struct MyClass {
    int Func(int const arg)
    {
        static std::atomic_flag f = ATOMIC_FLAG_INIT;
        if ( f.test_and_set() ) return -1;
        int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
        f.clear();
        return retval;
    }
};
```

NOTE: Applying `noreentry_all_objects` to a member function is identical in effect to applying `noreentry` to a normal function – however the simple keyword `noreentry` cannot be applied to a member function so that the programmer's intention is obvious at a glance.

The re-entry of a member function **for *all* objects** can be skipped for the same thread, and made to wait for other threads with the following syntax:

```
struct MyClass {
    int Func(int const arg)
    noreentry_all_objects(prevent_same:return -1,prevent_others:wait)
    {
        return (arg >= 0)? (arg + 7) : Func(-arg);
    }
};
```

and it behaves the same as:

```
struct MyClass {
    int Func(int const arg)
    {
        static std::recursive_mutex m;
        std::unique_lock<std::recursive_mutex> mylock(m);
        thread_local bool already_entered_by_same_thread = false;
        if ( already_entered_by_same_thread ) return -1;
        already_entered_by_same_thread = true;
        int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
        already_entered_by_same_thread = false;
        return retval;
    }
};
```

The re-entry of a member function **for the same object** can be skipped for the same thread, and made to wait for other threads with the following syntax:

```
struct MyClass {
    int Func(int const arg)
    noreentry_this_object(prevent_same:return -1,prevent_others:wait)
    {
        return (arg >= 0)? (arg + 7) : Func(-arg);
    }
};
```

And it behaves the same as:

```
struct MyClass {
    std::recursive_mutex m;
    std::set<std::thread::id> already_entered;
    int Func(int const arg)
    {
        std::unique_lock<std::recursive_mutex> mylock(m);
        if ( already_entered.contains(std::this_thread::get_id()) ) return -1;
        already_entered.insert(std::this_thread::get_id());
        int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
        already_entered.erase(std::this_thread::get_id());
        return retval;
    }
};
```

In total there are 10 unique schemes possible for preventing the re-entry of a member function:

(1) By the same thread, but only for `*this`, and always skip
(2) By the same thread, for *all* objects, and always skip
(3) By other threads, but only for `*this`, and always skip
(4) By other threads, but only for `*this`, and always wait
(5) By other threads, for *all* objects, and always skip
(6) By other threads, for *all* objects, and always wait
(7) By any thread, but only for `*this`, and always skip
(8) By any thread, for *all* objects, and always skip [see page 4]
(9) By any thread, for *all* objects, skip for same thread, wait for other threads [see page 5]
(10) By any thread, but only for `*this`, skip for same thread, wait for other threads [see page 6]

Two two keywords `noreentry_this_object` and `noreentry_all_objects` can be applied simultaneously to the same member function to produce a more elaborate scheme for preventing re-entry. In the following code snippet, the re-entry of a member function **for the same object** can be skipped for the same thread, and **for all other objects** can be made to wait for other threads, with the following syntax:

```
struct MyClass {
    int Func(int const arg)
    noreentry_this_object(prevent_same:return -1)
    noreentry_all_objects(prevent_others:wait)
    {
        return (arg >= 0)? (arg + 7) : Func(-arg);
    }
};
```

which behaves the same as:

```
struct MyClass {
    std::recursive_mutex mtx_for_same_object;
    std::set<std::thread::id> already_entered;
    int Func(int const arg)
    {
        std::lock_guard<std::recursive_mutex> lock_same_object(mtx_same_object);
        if ( already_entered.contains(std::this_thread::get_id()) ) return -1;
        static std::recursive_mutex mtx_for_all_objects;
        std::lock_guard<std::recursive_mutex> lock_all_objects(mtx_all_objects);
        already_entered.insert(std::this_thread::get_id());
        int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
        already_entered.erase(std::this_thread::get_id());
        return retval;
    }
};
```

In all of the sample implementations provided so far in this paper for preventing the re-entry of a member function, the object has been rendered unmovable and uncopyable because the two types, `std::recursive_mutex` and `std::atomic_flag`, are both unmovable and uncopyable. This restriction can be overcome by specifying an allocator when applying `noreentry_this_object`. The previous example on *Page 7* can be made movable and copyable as follows:

```
struct MyClass {
    int Func(int const arg)
    noreentry_this_object<std::allocator>(prevent_other_threads:return -1)
    noreentry_all_objects(prevent_other_threads:wait)
    {
        return (arg >= 0)? (arg + 7) : Func2(-arg);
    }
};
```

and it behaves the same as:

```
template<template<typename> class Alloc, typename T>
class shared_ptr_custom_alloc {
    static void alloc_deleter(T *const arg)
        noexcept( noexcept(arg->~T()) && noexcept(Alloc<T>().deallocate(arg,1u)) )
    {
        arg->~T();
        Alloc<T>().deallocate(arg,1u);
    }

    std::shared_ptr<T> p;

    void create_if_necessary(void) noexcept
    {
        if ( nullptr != p ) return;

        try
        {
            T *const q = Alloc<T>().allocate(1u);
            ::new(q) T();
            p.reset(q, &alloc_deleter, Alloc<T>());
        }
        catch(...) { std::abort(); }  // This is fatal -- must kill the process
    }

public:

    T &operator* (void) noexcept { create_if_necessary(); return *p;       }
    T *operator->(void) noexcept { create_if_necessary(); return p.get(); }
};
```

```
struct MyClass {
    template<typename T> using Alloc = std::allocator<T>;
    shared_ptr_custom_alloc< Alloc, std::recursive_mutex      > p_mtx_for_same_object;
    shared_ptr_custom_alloc< Alloc, std::set<std::thread::id> > p_already_entered    ;
    int Func(int const arg)
    {
        using std::recursive_mutex;
        std::lock_guard<recursive_mutex> lock_for_same_object(*p_mtx_for_same_object);
        if ( p_already_entered->contains(std::this_thread::get_id()) ) return -1;
        static recursive_mutex mtx_for_all_objects;
        std::lock_guard<recursive_mutex> lock_for_all_objects(mtx_for_all_objects);
        p_already_entered->insert(std::this_thread::get_id());
        int const retval = (arg >= 0)? (arg + 7) : Func(-arg);
        p_already_entered->erase(std::this_thread::get_id());
        return retval;
    }
};
```

When you move a `MyClass` object, the `shared_ptr` is simply moved, however when you copy a `MyClass` object, there are two possibilities:

(1) The `shared_ptr` is copied, meaning that the new `MyClass` object is sharing the same `recursive_mutex` and `atomic_flag` as the original `MyClass` object. This allows us to link a few different `MyClass` objects together in an elaborate scheme.

(2) A new `shared_ptr` is created holding a `nullptr`, meaning that the new `MyClass` object has its very own `recursive_mutex` and `atomic_flag` – so the newly created `MyClass` object is *not* linked to the original object.

The default behaviour when copying an object is No. 2:

```
MyClass a;
MyClass b(a);  // 'b' is not linked to 'a'
```

If you wish to create a link between `a` and `b`, you need to explicitly use a keyword for that:

```
MyClass a;
MyClass b(noreentry_linked_to a);  // 'b' shares mutexes with 'a'
```

Alternatively, you can designate an object as implicitly linking, as follows:

```
MyClass auto_propagate_noreentry_link a;
MyClass b(a);  // 'b' shares mutexes with 'a'
MyClass c(b);  // 'c' shares mutexes with 'b' and 'a'
```

The implicit linking can be turned on and off atomically:

```
MyClass auto_propagate_noreentry_link a;
MyClass b(a);  // 'b' shares mutexes with 'a'
set_propagate_noreentry_link b;
MyClass c(b);  // 'c' is not linked to 'b'
unset_propagate_noreentry_link c;
MyClass d(c);  // 'd' shares mutexes with 'c'
```

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/2023/02/5799.php