Exhaustive Dereferencing (`xdref`)

by T P K Healy

In C++23, we can already make chains out of invocations of '`operator->`' on objects of user-defined classes, for example:

```
#include <string>  // string

class B;

struct A {
    B  &operator->(void);
};

class C;

struct B {
    C  &operator->(void);
};

struct C {
    std::string *operator->(void) { static std::string s; return &s; }
};

B &A::operator->(void) { static B b; return b; }
C &B::operator->(void) { static C c; return c; }

#include <iostream>  // cout, endl

int main(void)
{
    A obj;

    obj->insert(0u, "World");
    obj->insert(0u, "Hello ");

    std::cout << obj->c_str() << std::endl;
}
```

Originally I was thinking that it would be useful if this '*chaining*' could also be applied to normal pointers, take the following code for example:

```
1: string s;
2: string    *p    = &s  ;
3: string   **pp   = &p  ;
4: string  ***ppp  = &pp ;
5: string ****pppp = &ppp;
6:
7: (***pppp)->insert(0u, "hello world");
```

It would be convenient if we could replace **Line #7** with the following:

```
pppp->insert(0u, "hello world");
```

If future C++ compilers were to allow this syntax, it would not break previous code because "`pppp->`" is currently a syntax error. It could however have implications for SFINAE, as shown by the following C++11-compliant snippet:

```
#include <utility>  // declval

template <typename T, typename Enable = void>
struct HasSize { static bool constexpr value = false; };

template<class...> using void_t = void;

template <typename T>
struct HasSize<T, void_t<decltype(std::declval<T>()->size())> > {
    static bool constexpr value = true;
};

#include <string>

int main(void)
{
    static_assert( HasSize<std::string**>::value );
}
```

The '`static_assert`' on the second-last line would fail in C++23, however it would succeed in future standards of C++ if chaining of normal pointers were allowed. It is important to note though that ***any*** change to the C++ core language would have an effect on SFINAE.

The convenience of writing:

```
pppp->insert(0u, "hello world");
```

instead of:

```
(***pppp)->insert(0u, "hello world");
```

comes with the following caveat:

Each of the four pointers, i.e. (`p`,`pp`,`ppp`,`pppp`), must not be a nullptr

The safe and proper way of using '`pppp`' would be:

```
if ( pppp && *pppp && **pppp && ***pppp ) pppp->insert(0u, "hello world");
```

While attempting to fathom all of the unintended consequences of allowing the implicit chaining of pointer dereferencing, I came up with the alternative idea of adding a new function called '`xdref`' to the standard library which can exhaustively dereference a pointer, or for that matter, any object that has an implementation of `operator->` or `operator*(void)`. The function '`xdref`' would keep on applying the operators `->` or `*` to an object until they cannot be applied anymore. '`xdref`' always returns an Lvalue (even if the supplied argument is an Rvalue). Here is an example of how '`xdref`' could be used:

```
using std::string; using std::unique_ptr; using std::optional;
string a, b, c, d, e;
string *p = &a;
string **pp0 = new string*(&b), **pp1 = new string*(&c);
unique_ptr<string> k(new string);
unique_ptr<string*> m(pp0), n(pp1);
unique_ptr<string*> *pm = std::addressof(n);
optional<string> os; os.emplace();
unique_ptr< optional<string*>* > puo(
    new optional<string*>*(new optional<string*>(new string)));

xdref(p).insert(0u,"K");
xdref(pp0).insert(0u,"K");
xdref(k).insert(0u,"K");
xdref(pm).insert(0u,"K");
xdref(os).insert(0u,"K");
xdref(puo).insert(0u,"K");
```

Each of those last six lines exhaustively dereferences the object until it can't be dereferenced anymore, in each case leaving us with an '`std::string`' object upon which we can invoke the '`insert`' method. If a **nullptr** is encountered, an exception will be thrown.

The implementation of ‘xdref’ in C++23 isn’t very complicated thanks primarily to two main features: ‘if constexpr’ (C++17) and ‘requires {}’ (C++20):

```
#include <type_traits>  // is_pointer_v, remove_reference_t
#include <utility>      // forward
#include <exception>    // exception

struct xdref_nullptr_exception : std::exception {
    char const *what(void) const noexcept override
    {
        return "Library function 'xdref' dereferenced a nullptr";
    }
};

template<typename T>
auto &&xdref(T &&p) noexcept(false) // p is either an Rvalue or Lvalue ref
{
    if constexpr ( std::is_pointer_v< std::remove_reference_t<T> > )
    {
        if ( nullptr == p ) throw xdref_nullptr_exception();

        // The next line recursively calls this function with ‘T’ as an Lvalue
        // because the dereferencing of a pointer always yields an Lvalue
        return xdref(*p);
    }
    else
    {
        if constexpr ( requires { std::forward<T>(p).operator->(); } )
        {
            return xdref( std::forward<T>(p).operator->() );
        }
        else if constexpr ( requires { std::forward<T>(p).operator*(); } )
        {
            return xdref( std::forward<T>(p).operator*() );
        }
        else
        {
            return p;  // Always returns an Lvalue without ‘forward’
        }
    }
}
```

Furthermore, since so many programmers and IT firms are still abiding the C++11 standard, I have implemented ‘`xdref`’ in C++11 as follows:

```
#include <type_traits>  // is_pointer, remove_pointer, remove_reference, enable_if
#include <utility>      // forward, declval
#include <exception>    // exception

struct xdref_nullptr_exception : std::exception {
    char const *what(void) const noexcept override
    {
        return "Standard library function 'xdref' dereferenced a nullptr";
    }
};

namespace xdref_detail {

template<typename T>
using rm_ref = std::remove_reference<T>;

template<typename T>
struct is_ptr { // Detect if any type is a pointer
    static bool constexpr value =
        std::is_pointer<typename rm_ref<T>::type>::value;
};

template<typename T>
struct rm_ptr { // Remove pointer from a pointer type
    typedef typename std::remove_pointer<typename rm_ref<T>::type>::type type;
};

template<typename T>
struct is_ptr2ptr { // Detect if any type is a pointer to a pointer
    static bool constexpr value =
          is_ptr<T>::value
      &&  is_ptr<typename rm_ptr<T>::type>::value
      && !std::is_same<typename rm_ref<T>::type, typename rm_ptr<T>::type>::value;
};
```

```
// The next 5 things are needed for SFINAE to detect operator-> and operator*(void)
template<class...> using void_t = void;  // Not in standard library until C++17

template<typename T, typename Enable = void>
struct HasOperatorArrow { static bool constexpr value = false; };

template<typename T>  // T might be 'T&' or 'T&&'
struct HasOperatorArrow<T, void_t<decltype(std::declval<T&&>().operator->())> > {
    static bool constexpr value = true;
};

template<typename T, typename Enable = void>
struct HasOperatorDeref { static bool constexpr value = false; };

template<typename T>  // T might be 'T&' or 'T&&'
struct HasOperatorDeref<T, void_t<decltype(std::declval<T&&>().operator*())> > {
    static bool constexpr value = true;
};

// Scenario 1 of 6: Not a pointer and cannot be dereferenced
template<
  typename T,
  typename std::enable_if<
       !is_ptr<T>::value
    && !HasOperatorArrow<T&&>::value
    && !HasOperatorDeref<T&&>::value,
    bool
  >::type = true
>
auto xdref(T &&p) noexcept -> T&  // Always returns an Lvalue
{
    return p;  // Always returns an Lvalue
}
```

```
// Scenario 2 of 6: A pointer (but not a pointer to a pointer)
//                  to an undereferenceable object
template<
  typename T,
  typename std::enable_if<
        is_ptr<T>::value
    && !is_ptr2ptr<T>::value
    && !HasOperatorArrow<typename rm_ptr<T>::type&>::value
    && !HasOperatorDeref<typename rm_ptr<T>::type&>::value,
    bool
  >::type = true
>
auto xdref(T &&p) noexcept(false) -> decltype(*p) & // Returns an Lvalue ref
{
    if ( nullptr == p ) throw xdref_nullptr_exception();

    return *p;  // Dereferencing a pointer always yields an Lvalue
}

// Scenario 3 of 6: A pointer to a pointer
template<
  typename T,
  typename std::enable_if<
    is_ptr2ptr<T>::value,
    bool
  >::type = true
>
auto xdref(T &&p) noexcept(false) -> decltype( xdref(*p) ) &&
{
    if ( nullptr == p ) throw xdref_nullptr_exception();

    return xdref(*p);  // Dereferencing a pointer always yields an Lvalue
}
```

```
// Scenario 4 of 6: Not a pointer but it has operator->
template<
  typename T,
  typename std::enable_if<
       !is_ptr<T>::value
    &&  HasOperatorArrow<T&&>::value,
    bool
  >::type = true
>
auto xdref(T &&p) noexcept(noexcept(std::forward<T>(p).operator->()))
  -> decltype( xdref(std::forward<T>(p).operator->()) ) &&
{
    return xdref( std::forward<T>(p).operator->() );
}

// Scenario 5 of 6: Not a pointer and doesn't have operator-> but has operator*
template<
  typename T,
  typename std::enable_if<
       !is_ptr<T>::value
    && !HasOperatorArrow<T&&>::value
    &&  HasOperatorDeref<T&&>::value,
    bool
  >::type = true
>
auto xdref(T &&p) noexcept(noexcept(std::forward<T>(p).operator*()))
  -> decltype( xdref(std::forward<T>(p).operator*()) ) &&
{
    return xdref( std::forward<T>(p).operator*() );
}
```

```
// Scenario 6 of 6: A pointer (but not a pointer to a pointer)
//                  to a dereferenceable object
template<
  typename T,
  typename std::enable_if<
        is_ptr<T>::value
    && !is_ptr2ptr<T>::value
    && (   HasOperatorArrow<typename rm_ptr<T>::type&>::value
        || HasOperatorDeref<typename rm_ptr<T>::type&>::value),
    bool
  >::type = true
>
auto xdref(T &&p) noexcept(false) -> decltype( xdref(*p) ) &&
{
    if ( nullptr == p ) throw xdref_nullptr_exception();

    return xdref(*p);  // Dereferencing a pointer always yields an Lvalue
}

}  // close namespace xdref_detail

template<typename T>
auto xdref(T &&p) noexcept(noexcept(xdref_detail::xdref( std::forward<T>(p))))
  -> decltype(xdref_detail::xdref(std::forward<T>(p))) &&
{
    return xdref_detail::xdref( std::forward<T>(p) );
}
```

Both the C++23 and C++11 implementations of ‘`xdref`’ work properly for all of the examples I’ve given so far, however I’ve been able to come up with a code snippet that works fine with the C++23 implementation, but won’t compile with the C++11 implementation:

```
#include <string>  // string
struct A; struct B; struct C;
struct A { B &&operator->(void) &&; };
struct B { C &&operator->(void) &&; };
struct C { std::string *operator->(void) { return new std::string; } };
int main(void) { xdref( A() ); }
```

The compiler complains of an ambiguous overload here for ‘`xdref`’ but I don’t understand why. If anyone can figure out what’s wrong I’d appreciate your help.