P3298R0

Implicit user-defined conversion functions

Bengt Gustafsson

St Louis – June 2024

Presentation contents

- Rationale
- History
- Proposed solution
- Resulting behavior
- Design decisions
- Examples

Rationale

- operator.() has been a requested feature for very long.
- To make proxy objects work as their proxied objects requires more than operator.() *seems* to provide:

- Costless conversion to the proxied object.*
- Using the proxied type as function parameter.*
- Using nested types and variables of the proxied type.
- Casting pointer to proxy to pointer to proxied object.

* The fact that N0416 does provide this does not change the expectations.

Rationale

- Inheritance offers all desired properties above
- Reusing the name lookup rules of inheritance simplifies reasoning
- Representing this as an *implicit* conversion function offers a logical place to implement the logic and is intuitive.

History

• P0416R0	Latest actual operator.() proposal, 2016
• P0352R0	First attempt to reuse inheritance, 2016
• P0700R0	Rebuttal of P0352 (with dubious claims).
• N4035	Complementary proposal to avoid dangling references. Needs an update.

Proposed solution

A conversion function declared **implicit** allows name lookup to be done as if the type inherited the return type of the conversion function.

```
template<typename T> struct Proxy {
    Proxy(T& object) : m_ptr(&object) {}

    implicit operator T&() { return *m_ptr; }
    implicit operator const T&() const { return *m_ptr; }

private:
    T* m_ptr;
};
```

```
struct MyClass {
    using Type = int;
    int x;
    void f();
    static void s();
};

void g(MyClass& o);

    MyClass obj;
    Proxy<MyClass> p(obj);

    p.f();                      // As Proxy<T> does not have an f check its bases and ICF return types
    p.x = 43;                   // As Proxy<T> does not have an x check its bases and ICF return types
    g(p);                       // As g does not take a Proxy<T> check its bases and ICF return types

    // All name lookup considers names in bases and ICF return types
    typename Proxy<MyClass>::Type anInt;        // Not needed with P2669

    // operator-> considers names in bases and ICF return types
    Proxy<MyClass>* pp = &p;
    pp->f();
    pp->MyClass::f();

    Proxy<MyClass>::s();        // Call static method of MyClass using name lookup.

    Base* bp = p;               // pointer to proxy can be converted to pointer to proxied if it is returned by reference.
```

```
template<typename T> struct Proxy {
    Proxy(T& object) : m_ptr(&object) {}

    implicit operator T&() { return *m_ptr; }
    implicit operator const T&() const { return *m_ptr; }

private:
    T* m_ptr;
};
```

Nomenclature

To reason about this we introduce the terms and abbreviations:

ICF:	Implicit conversion function
Handle:	The class containing an ICF.
Value:	The type returned by an ICF.

Obvious results

- Members in Value type found unless hidden by Handle members, as if Handle inherited from Value.
- Calls ICF to convert Handle to Value when needed.
- Access static members and types of Value using ::
- Use hidden members of Value by qualification with ::
- Works equally when -> is applied to Handle*.

Less obvious results

- Multiple levels of ICFs are called when needed.
- Inheritance and ICFs mix as multiple inheritance.
- ICFs returning subclasses need recursion avoidance during compilation.
- Pointer conversion can cause dangling, *forbidden*.
- **Implicit** is only a reserved word when followed by **operator**.
- Use **static_cast** to call hidden *virtual* member function.
- Virtual methods can’t be overridden in proxy.
- No downcasting from Value to Handle.

Design decisions

- Any type can be returned from an implicit conversion function, including fundamental types, final classes and array references.
- Virtual bases not accessible if there are two subobjects.
- Member pointers can work but are cumbersome.
- Incomplete and nested classes can be returned by ICFs.
- **sizeof**, **alignof** of Handle is independent of Value type.
- Handle can not access protected members in Value.
- ICFs can be virtual.

Examples

- Proxy-references: vector<bool>, simd, f-literals. *
- Lazy wrapper to use when value is maybe needed.
- Non-nullable smart pointers (aka smart references).

* Works best with N4035++. (using auto = T;)

Example: simd element reference

```
template<typename T, typename Abi> class simd {
public:
    struct reference {
        using auto = T;                    // N4035++
        using auto& = reference;     // N4035++

        reference(simd& s, int ix) : m_simd(s), m_ix(ix), m_val(m_simd.get(ix)) {}
        ~reference() { m_simd.set(m_ix, m_val); }
        implicit operator T& { return m_val; }

        simd& m_simd; int m_ix; T m_val;
    };
}

simd<float> x;

x[3] += 3.14f;                  // Works. += is done on float.

auto third = x[3];              // third is a float.
third *= 2;                     // does not affect x

auto& first = x[1];             // first is a simd<float>::reference
first -= 2.717f;                // This updates x[1]
```

Example: f-literals without performance loss

```
struct formatted_string {
    using auto = std::string;                    // N4035

    formatted_string(std::basic_format_string<char, Args...> fmt, Args&&... args) :
        m_fmt(fmt), m_args(std::make_format_args(std::forward<Args>(args)...)) {}

    implicit operator std::string() { return std::vformat(m_fmt.get(), m_args);

    std::basic_format_string<CharT, Args...> m_fmt;
    decltype(std::make_format_args(std::declval<Args>()...)) m_args;
};

int a = 17;
auto s = f"Value is {a}";                        // Here vformat runs to produce a std::string
std::println(f"Value is {a}");                   // Here a new println overload uses the members to optimize.
```

Example: lazy argument type

```
template<typename F> struct lazy {
    lazy(F f) : m_func(std::move(f)) {}

    implicit operator auto&() {
        if (!m_value)
            m_value = m_func();
        return *m_value;
    }


    F m_func;
    optional<decltype(func())> m_value;
};

void runIf(auto obj) {                  // Function unaware of Lazy arguments
    if (unlikely_event())
        obj.raise_alarm();              // obj only created if the alarm has to be sounded.
}

Lazy pp = &createObjectSlowly;          // With P3312 the function can be overloaded or a ctor.

runIf(pp);                              // Calls createObjectSlowly and raise_alarm only if needed.
```

Example: smart references

```
template<typename PTR> class universal_ref {
public:
    using value_type = pointer_traits<PTR>::element_type;

    universal_ref() = default;

     // Construct from the pointer-like, which must not be null.
    universal_ref(const PTR& src) pre (src) : m_ptr(src) {}
    universal_ref(PTR&& src) pre (src) : m_ptr(std::move(src)) {}

    // These conversions implement the operator.() functionality:
    implicit operator value_type&() & { return *m_ptr; }
    implicit operator const value_type&() const & { return *m_ptr; }
    implicit operator value_type() && { return std::move(*m_ptr); }        // Maybe not for shared_ptr!

    friend const PTR& unwrap(const universal_ref& src) { return src.m_ptr; }
    friend PTR unwrap(universal_ref&& src) { return std::move(src.m_ptr); }

private:
    PTR m_ptr;
};
```

Example: smart references

Many aspecs to consider for universal_ref.

- Maybe disallow move of universal_ref to avoid empty state. This makes unique_ptr specializations unmovable.
- Then users must do unwrap(std::move(src)) to get unique_ptr.
- Please don’t standardize std::polymorphic and std::indirect as ”pseudo references”. Standardize as cloning_ptr and add wrapper.