2020-09-16

The Many Shades of `reference_wrapper`

Zhihao Yuan <zy@simplerose.com>, SimpleRose Inc

HPC Engineer

How to switch between two objects?

Python:

```
dialog = some_object
if cond:
    dialog = some_other_object

process(dialog)
```

How to switch between two objects?

What about C++?

```
auto& dialog = some_object;
if (cond)
    dialog = some_other_object;  // ??

process(dialog);
```

C++ reference can only be bound once

```
auto& dialog = [&]() -> auto&
{
    if (cond) return some_object;
    else      return some_other_object;
}();
```

Pointers give you everything

```
auto *pdialog = &some_object;
if (cond)
    pdialog = &some_other_object;

process(*pdialog);
```

We ended up with…

```
unique_ptr<dialog_type> pdialog(new auto(some_object));
if (cond)
    pdialog.reset(new auto(some_other_object));

process(*pdialog);
```

Let’s look back a little bit…

Reference “assignment”

Python

```
dialog = some_object
if cond:
    dialog = some_other_object
```

C++

```
auto& dialog = some_object;
if (cond)
    dialog = some_other_object;
```

C++ reference

- initialization binds the object to the reference
- assignment assigns to the bounded object (so called “assign-through”)

What if initialization and assignment both bind objects?

- initialization binds the object to the reference
- assignment **rebinds** a different object to the reference
- like Python

std::reference_wrapper<T>

```
reference_wrapper dialog = some_object;
if (cond)
    dialog = some_other_object;

process(dialog);
```

`reference_wrapper` models

rebindable reference

`reference_wrapper` closely matches lvalue references

- it may refer to `const` objects
- it does not bind to rvalue expressions
- declaring `reference_wrapper<T>` does not odr-use `T`

Deduce from const-qualified lvalues

```
void f(vector<int> const& v)
{
    reference_wrapper r = v[0];   // reference_wrapper<int const>
}
```

Create from non-const lvalue

```
void f(vector<int> v)
{
    reference_wrapper r = std::as_const(v[0]);
}
```

C++20

reference_wrapper only binds to lvalue

```
reference_wrapper r = "foo"s;     // doesn’t compile
            auto& r = "foo"s;     // doesn’t compile
```

reference_wrapper<T const> only binds to lvalue, too

```
reference_wrapper<string const> r = "foo"s;  // nope
                  string const& r = "foo"s;  // okay
```

May refer to an incomplete type

```
class A;
void foo(A& x);
void bar(reference_wrapper<A> x);
```

C++20

Not only rebindable, but also assign-through

reference_wrapper<T> is convertible to T&

```
reference_wrapper dialog = some_object;
if (cond)
    dialog = some_other_object;

process(dialog);  // void process(dialog_type&);
```

You may also reach the bounded object with `.get()`

```
reference_wrapper dialog = some_object;
if (cond)
    dialog = some_other_object;

process(dialog.get());
```

Combine rebinding and assign-through

Bind object `b` to `r`:

```
reference_wrapper r = a;
r = b;
```

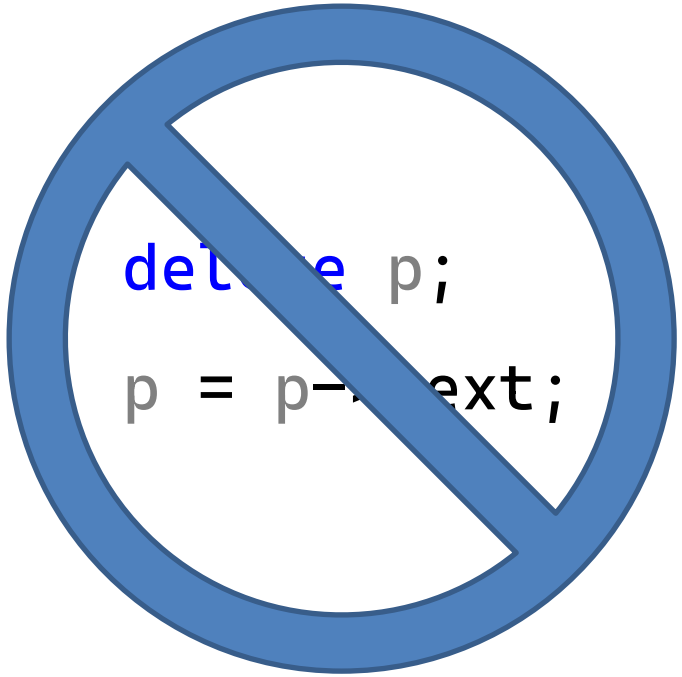
Assign `b` to `r`'s referenced object:

```
reference_wrapper r = a;
r.get() = b;
```

Let's talk about linked list

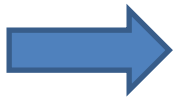
```
struct node { node *next = nullptr; string val; };


struct linked_list
{
    node *head;
    void remove(string_view val);  // how to implement?
};
```

A tiny little trick



std::exchange

```
auto saved = p;
p = p->next;
delete saved;
```



```
delete std::exchange(p, p->next);
```

C++14

Dropping node by changing `last->next`

```
for (node *last = nullptr, *p = head; p != nullptr;)
{
    if (p->val == val)
    {
        if (last)
            last->next = p->next;
        else
            head = p->next;
        delete std::exchange(p, p->next);
    }
```

(continued)

```
for (node *last = nullptr, *p = head; p != nullptr;)
{
    if (p->val == val)
    { /* … */ }
    else
    {
        last = p;
        p = p->next;
    }
}
```

Thoughts

- We were directly modifying `head`

  ```
  head = p->next;
  ```

  as if we have

  ```
  void remove_impl(node *&head, string_view);
  ```

- Can we form a reference to pointer (*e.g.*, `node *&`) for each current node?

Rebind the reference-to-head to other nodes

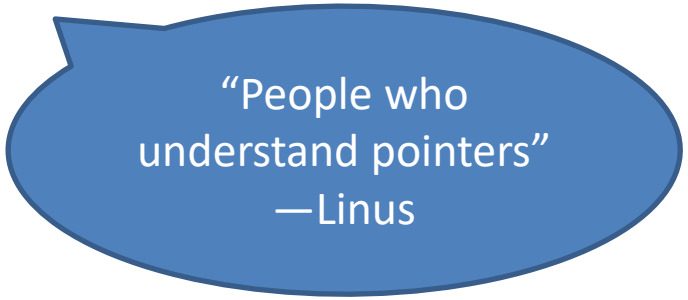
```
reference_wrapper p = head;
while (p != nullptr)
{
    if (p.get()->val == val)
        delete std::exchange(p.get(), p.get()->next);
    else
        p = p.get()->next;
}
```

`p.get() = p.get()->next;`
Relinking by replacing the current node with the next node

Iterating by rebinding the reference to the next node

“Two star programming”

```
for (node **pp = &head; *pp != nullptr;)
{
    node *entry = *pp;
    if (entry->val == val)
    {
        *pp = entry->next;
        delete entry;
    }
    else
        pp = &entry->next;
}
```



Pointers have very mixed semantics

The following material are stolen from Walter E. Brown.

- A pointer value is the value of a pointer variable:
  - Just like all variables' values, a pointer value is an rvalue.
  - Unlike rvalues of other types, a pointer value can be treated as an lvalue (*e.g.*, via unary operator *).
- A pointee is a variable whose lvalue corresponds to the rvalue of some pointer variable.
- A pointer value is simultaneously:
  - An lvalue (from the pointee's perspective), and…
  - An rvalue (from the pointer variable's perspective).

Exercise

```
struct linked_list
{
    node *head;
    // read from a file, one line per node
    void read(std::istream &fin);
};
```

Answer

```
string ln;
reference_wrapper ls = head;
while (getline(fin, ln))
{
    ls.get() = new node{ .val = ln };
    ls = ls.get()->next;
}
```

Last topic

You probably have seen code like this

```
auto f = some_function;
if (cond)
    f = some_other_function;
f(args);
```

Function pointers

– a rebindable reference in the language

- its usage models after references
  - you don't need to write `(*f)(args)`
  - just `f(args)`, as if `f` is a reference to function (*e.g.*, an object of type `void (&)(int)`)
- it's rebindable
  - via assignment

```
f = a_different_function;
```

If we replace auto with reference_wrapper…

```
auto f = some_function;
if (cond)
    f = some_other_function;
f(args);
```

reference_wrapper works as a function pointer as well

```
reference_wrapper f = some_function;
if (cond)
    f = some_other_function;
f(args);
```

operator()

Only better[1]

- `reference_wrapper` closely matches function pointer's capability
- it extends the scope of pointee to $Callable$ objects
- it's **not** nullable

[1]cannot be used as non-type template parameter yet

Works in constexpr, just like function pointers

```
constexpr int foo(bool cond)
{
    reference_wrapper f = int_f;
    if (cond)
        f = int_g;
    return f();
}

constexpr auto v = foo(true);
```

C++20

Properly constrained, behaves like function pointers

```
reference_wrapper f = atoi;


static_assert(std::is_invocable_v<decltype(atoi), char*>);
static_assert(std::is_invocable_v<decltype(f), char*>);
static_assert(!std::is_invocable_v<decltype(atoi), float>);
static_assert(!std::is_invocable_v<decltype(f), float>);
```

Callable objects, you mean pointer-to-members?

- `reference_wrapper` can bind to lvalue pointer-to-members and call them with the ordinary function call syntax
- but you cannot form a “reference to member” because pointer-to-member has no pointee; there is no such entity called “member”
- `std::mem_fn` is probably what you are looking for

Works with user-defined callable objects

```
template<class T>
struct plusN
{
    auto operator()(T const& x) const { return x + n; }
    T n;
};

plusN plus5{ 5 }, plus7{ 5 };
```

```
reference_wrapper fn = plus5;
                  fn = plus7;
```

It’s not omnipotence though

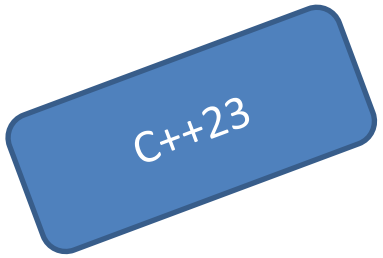
```
std::plus f;
std::multiplies g;
reference_wrapper fn = f;
fn = g;  // nope
```

We can ask `std::function` to erase the types, or…

```
std::plus f;
std::multiplies g;
function<int(int, int)> fn = reference_wrapper(f);
fn = reference_wrapper(g);
```

Use a type-erased rebindable reference to functions

```
std::plus f;
std::multiplies g;
function_ref<int(int, int)> fn = f;
fn = g;
```



What can replace function pointers?

`reference_wrapper<T>`

- non-null
- reference semantics
- `T` is pointee's type
- `sizeof(T*)`

`function_ref`

(as of P0792R5)

- nullable
- reference semantics
- type-erased
- at least 2x the size

`std::function`

- nullable
- value semantics
- type-erased
- only larger

Summary

- `reference_wrapper` is a rebindable reference
- `reference_wrapper` makes reference-binding and assign-through explicit
- `reference_wrapper` is a better function pointer

Questions

@lichray