A Proper Property

Gašper Ažman

November 27, 2017

About Me

- Gašper Ažman (twitter: @atomgalaxy)
- Started teaching C++ in highschool
- Did computer vision research at Berkeley
- Helped with Amazon Search Infrastructure at a9.com
- Currently at Citadel, building really cool research tools.
- A regular at the British Standards Insitute (BSI) Meetings
- On the programming committes of CppCon and C++Now.

Disclaimer

The opinions in this talk are my own, and do not necessarily reflect the opinions of Citadel LLC or any of its subsidiaries.

In addition, no Citadel resources were used in the development of this talk.

So, What is a Property?

A property pretends to be a field.

Assignment:

```
1 airplane.hold = "hot air";
```

Reading:

```
1 Payload x = airplane.hold;
```

(Aside: we need a payload, and strings do nicely.)

```
1 // books truly are the greatest gift
2 using Payload = std::string;
```

Totally.

If it quacks like a field, it has to be...

```
1 class Airplane {
2   Payload hold;
3 } airplane;
```

... a field, right?

Have you heard of this?

It is only a shell...

Instead, it's a pair of a Getter and a Setter on a member.

```
1  class Airplane {
2    struct Cargo {
3      // Setter - assignment from Payload
4      void operator=(Payload crate) {
5        return {};
6      }
7      // Getter - implicit conversion to Payload
8      operator Payload() const {
9        return {};
10     }
11   };
12   Cargo hold;
13 } airplane;
```

(We need at least one byte to give the `hold` an address).

Let's Get More Interesting


```
1 class Airplane {
2   bool hatch_closed;
3 public:
```

Let's Get More Interesting


```
1  class Airplane {
2    bool hatch_closed;
3  public:
4    struct Cargo {
5      Payload payload;
6      void operator=(Payload crate) {
7        if (hatch_closed) throw ClosedError{};
8        payload = std::move(crate);
9      }
```

Let's Get More Interesting


```
1  class Airplane {
2    bool hatch_closed;
3  public:
4    struct Cargo {
5      Payload payload;
6      void operator=(Payload crate) {
7        if (hatch_closed) throw ClosedError{};
8        payload = std::move(crate);
9      }
10     operator Payload const&() const {
11       if (hatch_closed) throw ClosedError{};
12       return payload;
13     }
```

Let's Get More Interesting


```
1  class Airplane {
2    bool hatch_closed;
3  public:
4    struct Cargo {
5      Payload payload;
6      void operator=(Payload crate) {
7        if (hatch_closed) throw ClosedError{};
8        payload = std::move(crate);
9      }
10     operator Payload const&() const {
11       if (hatch_closed) throw ClosedError{};
12       return payload;
13     }
14   } hold;
15 } airplane;
```

Let's Get More Interesting


```
1  class Airplane {
2    bool hatch_closed;
3  public:
4    struct Cargo {
5      Payload payload;
6      void operator=(Payload crate) {
7        if (hatch_closed) throw ClosedError{};
8        payload = std::move(crate);
9      }
10     operator Payload const&() const {
11       if (hatch_closed) throw ClosedError{};
12       return payload;
13     }
14   } hold;
15 } airplane;
```

Pro: this solution is pretty.

Let's Get More Interesting


```
1  class Airplane {
2    bool hatch_closed;
3  public:
4    struct Cargo {
5      Payload payload;
6      void operator=(Payload crate) {
7        if (hatch_closed) throw ClosedError{};
8        payload = std::move(crate);
9      }
10     operator Payload const&() const {
11       if (hatch_closed) throw ClosedError{};
12       return payload;
13     }
14   } hold;
15 } airplane;
```

Pro: this solution is pretty.

Con: it is not a solution. (It does not compile.)

But WHY?

We need (Airplane*) `this`

Not `(Cargo*) this`.

But... We wants it! We needs it, precious!

No.

You're not getting a second breakfast... I mean, a second `this`.

Get The Host's `this`.

... while being reasonably easy to use.

Attempt 1:

Keep It Surprisingly Simple

Store the this pointer


```
1  class Airplane {
2    bool hatch_closed;
3  public:
4    struct Cargo {
5      Payload payload;
6      void operator=(Payload crate) {
7        if (host->hatch_closed) throw ClosedError{};
8        payload = std::move(crate);
9      }
10     operator Payload const&()  const {
11       if (host->hatch_closed) throw ClosedError{};
12       return payload;
13     }
14     Airplane* host;
15   } hold;
16   Airplane() : hold{this} {} // every. time.
17 } airplane;
```

So... How'd we do?

Poorly.

Needs extra space? Check.

Error-prone? Check.

No help from C++? Check.

Easy? To understand, yes. To maintain? Good luck.
(does not pass the WWTDCD[1] test)

[1] What Would The Default Constructor Do

Moving the Goalposts Much?

We need some criteria.

Criteria

The First Rule of C++

Criteria

The First Rule of C++

We only pay for what we use.

Criteria

The First Rule of C++

We only pay for what we use.

At Most One Macro Per Property

The generated code must be contiguous.

Criteria

The First Rule of C++

We only pay for what we use.

At Most One Macro Per Property

The generated code must be contiguous.

No Pitfalls

- Easy to read
- Easy to write
- Easy to modify

Criteria

The First Rule of C++

We only pay for what we use.

At Most One Macro Per Property

The generated code must be contiguous.

No Pitfalls

- Easy to read
- Easy to write
- Easy to modify

Boils down to *Don't repeat yourself.*

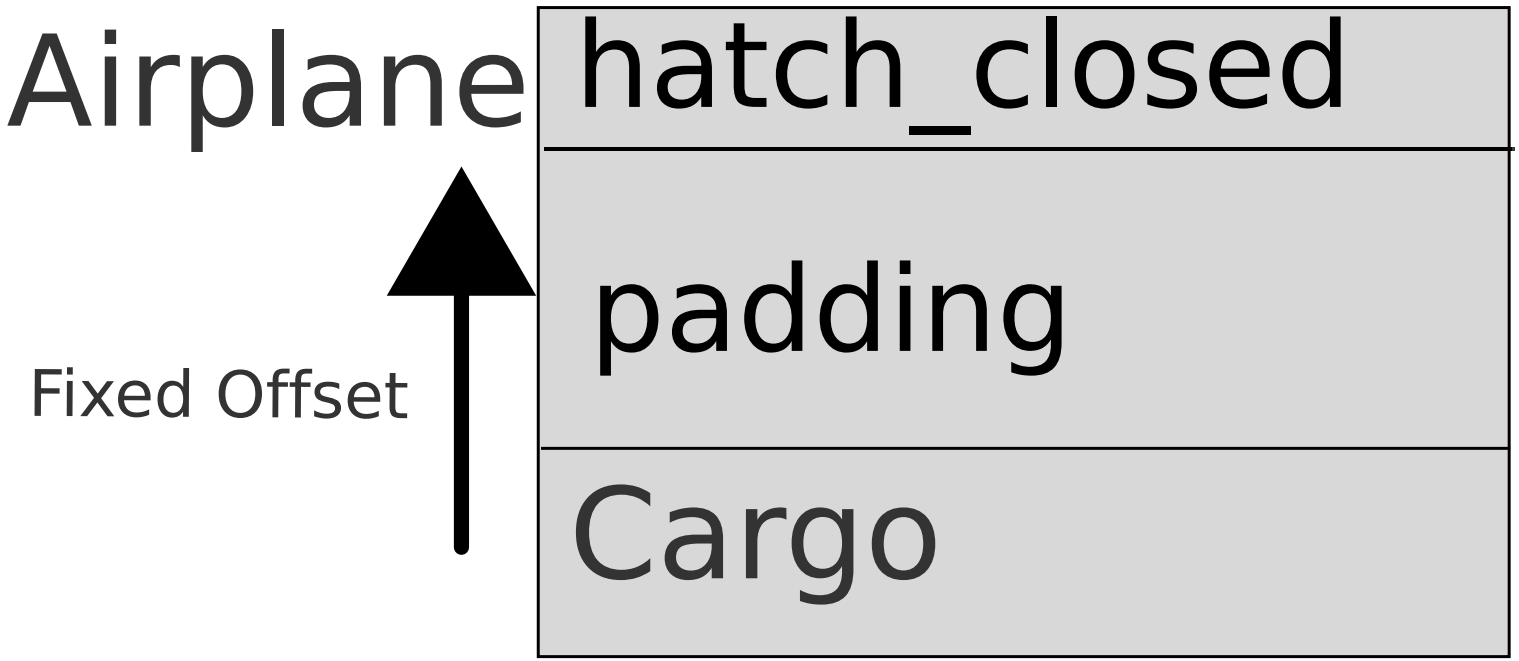
And we had to repeat ourselves with every constructor and assignment operator.

Attempt 2:

offsetof

Member Offsets are Constant

We already have `(Cargo*) this`. We can **compute** `(Airplane*) this`.



x86_64, clang: `&Airplane::hold` == 8

Easy Peasy!

We have:

```
1 //
2   auto offset = offsetof(Airplane, hold);
3   auto fixed  =
4                             this  - offset;
5   auto host   =                           fixed ;
6 //
```

Now For Something That Actually Works

With the casts in:

```
1 //
2   auto offset = offsetof(Airplane, hold);
3   auto fixed  =
4     reinterpret_cast<char*>(this) - offset;
5   auto host   = reinterpret_cast<Airplane*>(fixed);
6 //
```

If You Like It, Put It In A Function

With a function around it:

```
1 Airplane* get_host() {
2   auto offset = offsetof(Airplane, hold);
3   auto fixed  =
4     reinterpret_cast<char*>(this) - offset;
5   return        reinterpret_cast<Airplane*>(fixed);
6 }
```

But...

But, it has warnings!

```
offset_of.cpp:34:23: warning: offset of
  on non-standard-layout type 'Airplane'
  [-Winvalid-offsetof]

auto offset   = offsetof(Airplane, hold);
                ^                  ~~~~
```

(No, it's not UB, if you're using c++17)

But, Is it... Legal?

Turns out this doesn't have a warning, and is better than `constexpr`.

```
1  std::integral_constant<size_t,
2    offsetof(Airplane, cargo)>::value;
```

Why is this OK?

Because there can never be anything virtual between a member and its *directly enclosing class*.
Think of a non-`virtual` member function. It uses the `this` pointer and an offset to get to the member, because there is no vtable. That offset has to be constant.
What we are doing has no chance of not working.

So... How'd we do?

We did OK. For a once-off.

- The getter and setter pair are completely ad-hoc.
- Ad-Hoc `get_host()` function with scary casts.

How About This?


```
1  class Airplane {
2    struct Cargo {
3      Payload payload;
4      template <typename Host>
5      void set(Host& host, Payload crate) {
6        if (host.hatch_closed) raise ClosedError{};
7        payload = std::move(crate);
8      }
9      template <typename Host>
10     auto const& get(Host const& host) const {
11       if (host.hatch_closed) raise ClosedError{};
12       return payload;
13     }
14   };
15 public:
16   bool hatch_closed;
17   LIBPROPERTY_WRAP((Cargo), hold, Airplane);
18 };
```

The Anticlimax

I don't think we can get away without macros.

The Anticlimax

I don't think we can get away without macros.

But, I promise they're not the worst thing about this solution.

The Anticlimax

I don't think we can get away without macros.

But, I promise they're not the worst thing about this solution.

Wait, that's not a good thing.

The Anticlimax

I don't think we can get away without macros.

But, I promise they're not the worst thing about this solution.

Wait, that's not a good thing.

... well, maybe they are.

Down The Rabbit-Hole

With get_host(), we had:

```
1  class Airplane {
2    struct Cargo {
3      Payload payload;
4      void operator=(Payload crate) {
5        auto& host = *Airplane::get_host(this);
6        /* use host.hatch_closed */
7      }
8    };
9  public:
10   bool hatch_closed;
11   Cargo hold;
12
13   static Airplane* get_host(Cargo* cargo) {
14     return /* cargo - offsetof(Airplane, cargo); */
15   }
16 };
```

To Make A Macro

We need contiguous, repetitive, easily substitutable code.

Also, *sure, we had one cargo, but...*

But What About Second Cargo?

This is pretty doable:

```
class Airplane {
  template <auto closed>
  struct Cargo {
    Payload payload;
    void operator=(Payload crate) {
      auto const& host = Airplane::get_host(*this);
      if (host.*closed) throw ClosedError{};
      payload = std::move(crate);
    }
  };
public:
  bool hold_closed;
  bool frunk_closed;
  Cargo<&Airplane::hold_closed> hold;
  static Airplane& get_host(decltype(hold)&);
  Cargo<&Airplane::frunk_closed> frunk;
  static Airplane& get_host(decltype(frunk)&);
};
```

OK Now?

Better, but we still need to:

- manually get the host pointer.
- We need at least 3 overloads of `get_host: (const&, &, &&)` per member.
- We need to manually type `Airplane::`
- `get_host()` pollutes the interface of Airplane, and choosing an uglier and less-likely-to-clash name makes our implementation uglier too.
- What if we only had one hatch?

Store It In The Type

Attempt 3:

Store It In The Type.

But What About Second Cargo, Reprise?

What if there is only one hatch?

```
1  class Airplane {
2    template <auto closed>
3    struct Cargo { Payload payload; /* */ };
4  public:
5    bool hatch_closed;
6    Cargo<&Airplane::hatch_closed> hold;
7    Cargo<&Airplane::hatch_closed> frunk;
8    static Airplane& get_host(decltype(hold)&);
9    // ERROR: same type
10   static Airplane& get_host(decltype(frunk)&);
11 };
```

But What About Second Cargo, Reprise II?

Type tags to the rescue!

```
1  class Airplane {
2    template <auto closed, typename Tag>
3    struct Cargo { Payload payload; /* */ };
4  public:
5    bool hatch_closed;
6    struct hold_tag;
7    Cargo<&Airplane::hatch_closed, hold_tag> hold;
8    struct frunk_tag;
9    Cargo<&Airplane::hatch_closed, frunk_tag> frunk;
10   static Airplane& get_host(decltype(hold)&);
11   // Now fine.
12   static Airplane& get_host(decltype(frunk)&);
13 };
```

Type Tags Are Awesome?

But wait, there's more!

```
class Airplane {
  template <auto closed, typename Tag>
  struct Cargo { Payload payload; /* */ };
public:
  bool hatch_closed;
  struct hold_tag {
    static auto offset_of() {
      // Airplane is fully defined, unless it isn't
      // when *called*
      return std::integral_constant<size_t,
        offsetof(Airplane, hold)>{};
    }
  };
  Cargo<&Airplane::hatch_closed, hold_tag> hold;
  struct frunk_tag { /* same as above */ };
  Cargo<&Airplane::hatch_closed, frunk_tag> frunk;
};
```

In Other News, Stack Corruption.

Also, what about this?

```
1 Airplane airplane;
2 auto x = airplane.cargo; // works!
3 x = "foo";               // corrupts the stack
```

Fix


```
1  template <auto managed>
2  struct Cargo {
3    friend Airplane;
4    /* getters, setters */
5    private:
6    Cargo() = default;
7    Cargo(Cargo const&) = default;
8    Cargo(Cargo&&) = default;
9    Cargo const& operator=(Cargo const&) = default;
10   Cargo&& operator=(Cargo&&) = default;
11   ~Cargo() = default;
12 };
```

Now only Airplane can manage Cargo.

... Better.

Now Breaks:

```
1 Airplane airplane;
2 // breaks, copy constructor is private.
3 auto x = airplane.cargo;
```

Really Though?

This is a lot of code. We want to put this into a library.

It gets to be *a lot more code* when you want return-type deduction, SFINAE, and templates for getters and setters to work correctly.

So, What do We Need?

We need to wrap our getter/setter provider into an adaptor.

```
1  template <typename Property, typename Tag>
2  class wrapper {
3    // allow 'host' to access self::value
4    friend host;
5    Property value;
6
7    constexpr wrapper()=default;
8    constexpr wrapper(wrapper const&)=default;
9    constexpr wrapper(wrapper&&)=default;
10   ~wrapper()=default;
11   constexpr wrapper& operator=(wrapper const&)=
         default;
12   constexpr wrapper& operator=(wrapper&&)=default;
```

Setters


```
1  template <typename Property, typename Tag>
2  class wrapper { /*cont*/
3    /* setter implementation */
4    template <typename X>
5    decltype(auto) operator=(X&& val) & {
6      return value.set(
7           ::libproperty::impl::get_host(*this),
8             std::forward<X>(val));
9    }
10   /* and the const& and && variants */
```

Getters


```
template <typename Property, typename Tag>
class wrapper { /*cont*/
  // SFINAE-detect getter presence
  // Defer to call time by making type dependent
  // also: you must say it 3 times (Vittorio,
      thanks!)
  template <typename V = value_type, // fake params
            typename H = host,
            bool nxc = noexcept(
              std::declval<V>().get(
                std::declval<H const&>())))>
  auto get() const &
       noexcept(nxc) -> decltype(auto)
  {
    return value.get(
      ::libproperty::impl::get_host(*this));
  }
  /* and the & and && variants */
```

Implicit Conversions


```
1  template <typename Property, typename Tag>
2  class wrapper { /*cont*/
3    // SFINAE-detect get() presence...
4    // also: you must say it 3 times (Vittorio,
         thanks!)
5    /* and the & and && variants */
6    template <typename W = wrapper> // type-dependent
7    operator decltype(
8      std::declval<W const&>().get())() const &
9    {
10     return get();
11   }
```

The Magic Macro


```
#define LIBPROPERTY_WRAP(type, name, host) \
  LIBPROPERTY__DECLARE_TAG(name, host);    \
  ::libproperty::wrapper<                  \
      LIBPROPERTY__PARENTHESIZED_TYPE type,\
      host::LIBPROPERTY__TAG_NAME(name)>   \
      name;                                \
  static_assert("require semicolon")

#define LIBPROPERTY__DECLARE_TAG(name, host)   \
  struct _libproperty__##name##_prop_tag {     \
    using host_type = host;                    \
    auto static constexpr offset()             \
    {                                          \
      return std::integral_constant<size_t,    \
        offsetof(host, name)>{};               \
    }                                          \
  };                                           \
  static_assert("require semicolon")
```

That's it!

There are a few helpers to ferry data to-and-fro.

The trick is really in defining the tag type *outside* the property, so we can reuse our wrapper.

The *other* trick is doing the `offsetof` inside an `auto`-typed constxpr function that returns an integral constant.

This defers lookup until all the types of all data members are known.

So, This Works!

We have avoided many, many hours of "you can't do this because the type isn't defined yet" with this path.

Libproperty has a lot more cool features:

- You can get the host pointer as a universal reference, and thus only have to write one getter template.
- The host is specified once, in the macro.
- You can store the value *in* the Cargo object. We could do that here too, and avoided the space penalty, but there are pitfalls.
- If you want comparisons with strings to work, you need to overload all of them - the library forwards those for you.

Questions?

Thank You.

BONUS SLIDES

I promised casting zero to stupid things.

This code actually used to be in the library:

```
1 template <typename Host,
2   typename PointerToMemberType>
3 auto constexpr offset_of(
4     PointerToMemberType member_ptr) {
5   return reinterpret_cast<std::size_t>(
6     &((Host*)0->*member_ptr));
7 }
```

It works. It's not `constexpr`, but it works with partially defined types, and `offsetof` didn't use to before C++17 (at least in clang and gcc).