C++ Cryptozoology

A Compendium of Cryptic Characters :: #2

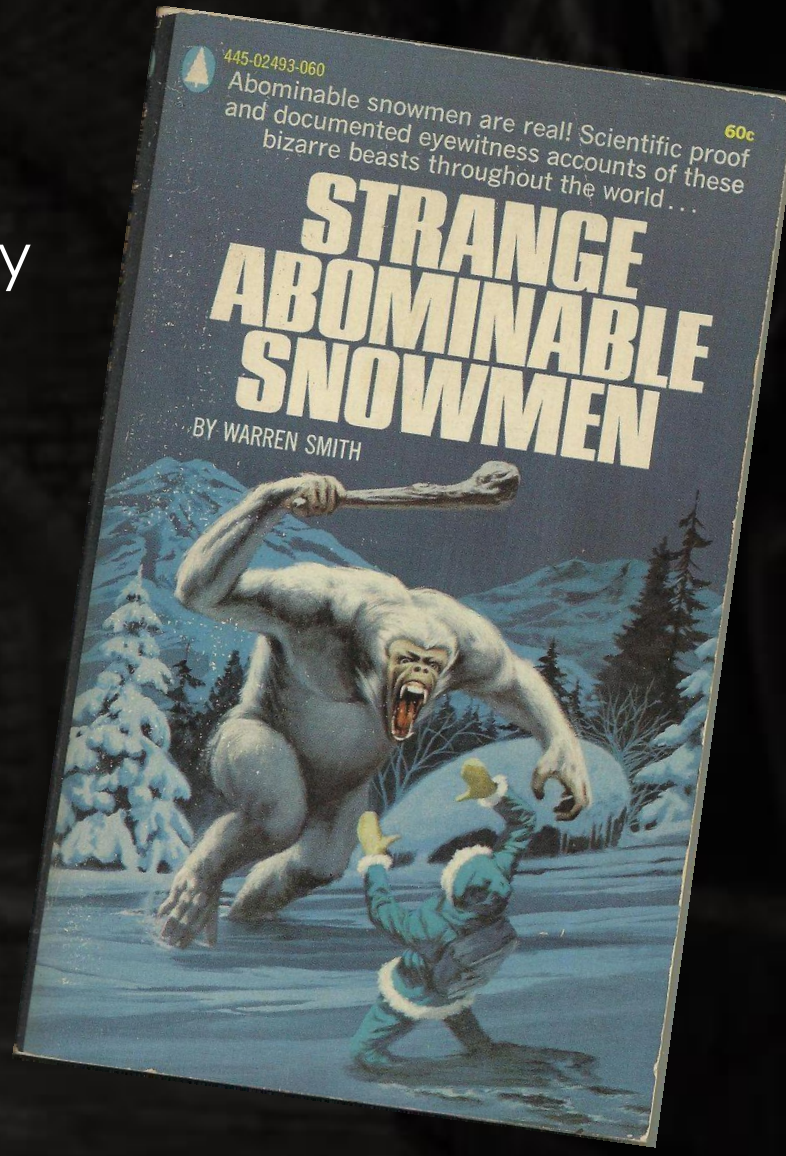
Abominable Function Types

An *abominable function type* is the type produced by writing a function type followed by a *cv-ref qualifier*:

```
using abominable = void() const volatile &&;
```

- `abominable` is a *function type*
- *NOT* a reference type, *NOT* `const` *NOT* `volatile` !

Impossible to create a function of abominable type!



"There is a dark corner of the type system that is little known other than to compiler writers..."
– Alisdair Meredith, Abominable Function Types

Abominable Function Types

```
struct rectangle
{
    using int_property = int() const;                    // common signature for several methods
    int_property top, left, bottom, right, width, height; // declare property methods!
    // ...                                                        ^^^^^^^
};
```

COWs

- **Copy-on-Write**: An object copy shares its internal state until modified.
- Popular for resource management in OSs
  - Virtual memory, file systems, databases
- 1999: *"...to my knowledge all of the most popular implementations of `basic_string` on Windows and other platforms use COW." – HS, CUJ1999*
- C++11 forbids COW for `std::string` due to iterator/pointer/reference invalidation.

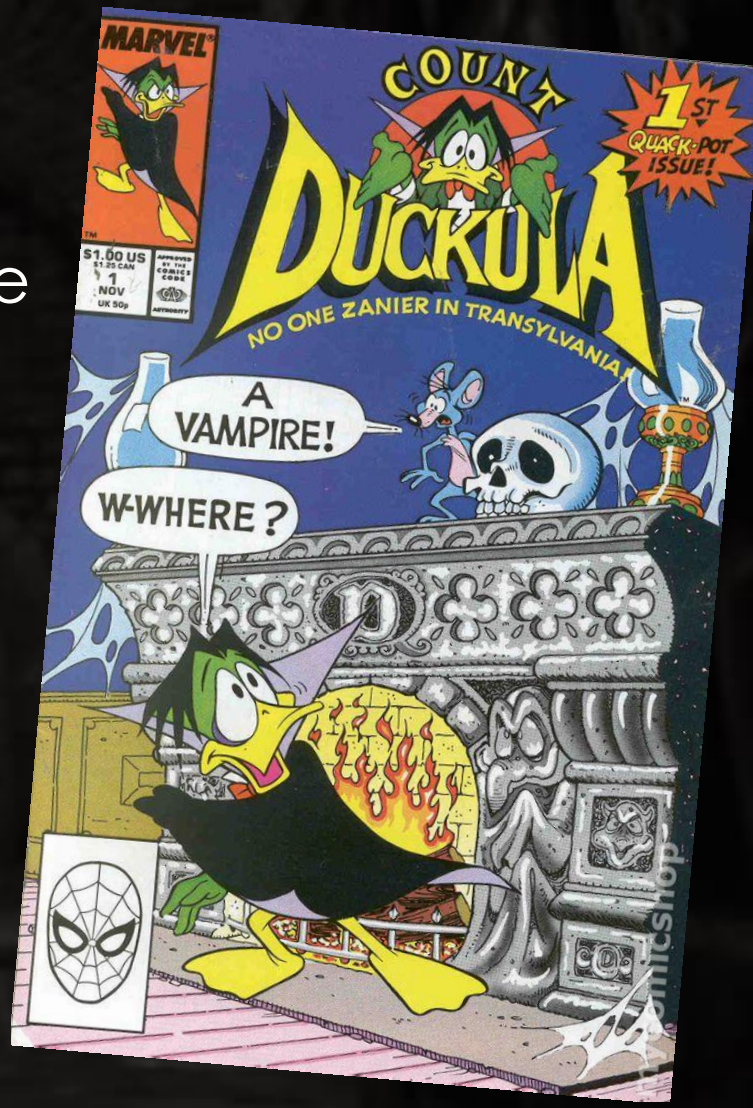
"...no call to `operator[]()` may invalidate pointers, references or iterators..."
– Johnathan Wakely, SO 2015

Duck Typing

- *Duck Typing* is an application of *The Duck Test* in type safety.
- A form of *abductive reasoning*:

  If it looks like a duck,
  swims like a duck, and
  quacks like a duck,
  then it probably is a duck.

- Classic form: At runtime → dynamically typed langs
- In C++ templates: Statically at *instantiation* point
- C++20 Concepts: template type specification:
  No Duck Typing!



"If it looks like a duck and quacks like a duck but it needs batteries, you probably have the wrong abstraction."
– The Internets on the Liskov Substitution Principle

Magical Code

Adi Shavit
@AdiShavit

C++ Gurus: How many examples can you name of magical C++ types/functions (eg initializer_lists, std::launder etc)? Both ISO or imp-def. are valid.

Is there a comprehensive list somewhere?

10:13 AM - 23 May 2018

Corentin @Cor3ntin · May 23
Placement new in mallocated memory is UB - works fine in practice obviously.
For std::vector the wording has to hand wave it :p

sbi @tweetsbi · May 24
Replying to @AdiShavit
Has anyone mentioned std::type_info yet?

Simon Brand @TartanLlama · May 23
Replying to @AdiShavit
A good half of the type traits

Shafik Yaghmour @shafikyaghmour · May 23
Replying to @AdiShavit
This is a little vague to me ... do builtins count?

__builtin_constant_p is pretty magical stackoverflow.com/a/24400015/170...

> +// __builtin_constant_p ? : is magical, and is always a potential constant.
>
> and:
>
> - // This macro forces its argument to be constant-folded, even if it's not
> - // otherwise a constant expression.
> - **define fold(x) (__builtin_constant_p(x) ? (x) : (x))**

Thomas Rodgers @rodgertq · May 23
Replying to @AdiShavit
Most of <atomic> is magic

std::hardware_destructive/constructive_interference_size are magic values

Victor Zverovich @vzverovich · May 23
Replying to @AdiShavit
More of a C than C++, but va_list and friends.

Matt Calabrese @CppSage · May 23
Replying to @AdiShavit
std::addressof

Dimitar Mirchev @DVMirchev · May 23
Replying to @AdiShavit
std::is_union is unimplementable without compiler support

Peter Bindels @dascandy42 · May 23
Replying to @AdiShavit
Typeinfo, vector, std bless, is_noexcept_destructible iirc.

Ben Deane @ben_deane · May 23
Replying to @AdiShavit
Magical function #1 is surely main :)

Casey Carter @CoderCasey · May 25
[complex.numbers]/4.4 and 4.5 are the requirements that aren't implementable in C++.

Casey Carter @CoderCasey · May 25
Replying to @AdiShavit
node_handle "magics" away a reinterpret cast that allows you to modify the key value in a pair<const Key, Value> after extracting it from a std::map. (eel.is/c++draft/conta...) which is the reason it forbids specializations of pair (eel.is/c++draft/conta...).

*"C++ *is* magic."*
– @davidbrcz

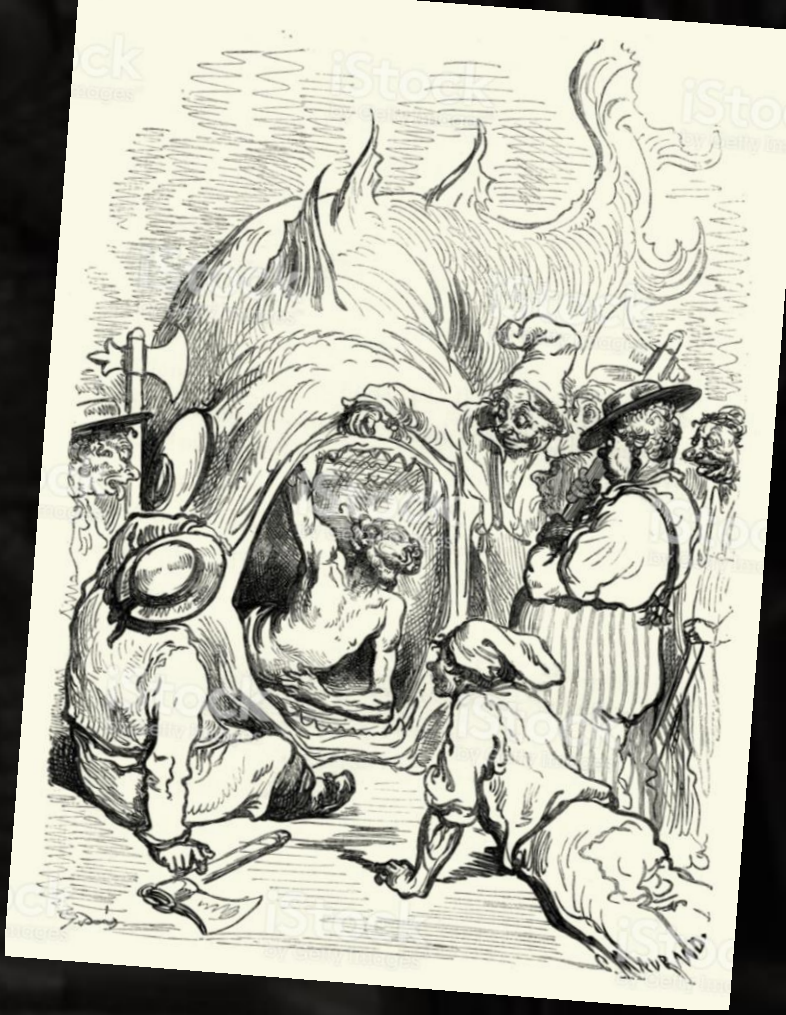
Maximal Munch

- When the lexical analyzer take as many characters as possible to form a valid token.

  "Otherwise, the next preprocessing token is the longest sequence of characters that could constitute a preprocessing token, even if that would cause further lexical analysis to fail." -- [lex.pptoken]

- So: `x++y` means:
  - Invalid code: `x++ y; // invalid`
  - ***Not*** the valid: `x+ +y;`
- The culprit for *historic* parsing traps like:

```
typedef std::vector<std::vector<int> > Table;  // OK
typedef std::vector<std::vector<bool>> Flags;  // Error
```



"Why doesn't `a+++++b` work?"
– Stumped SO User

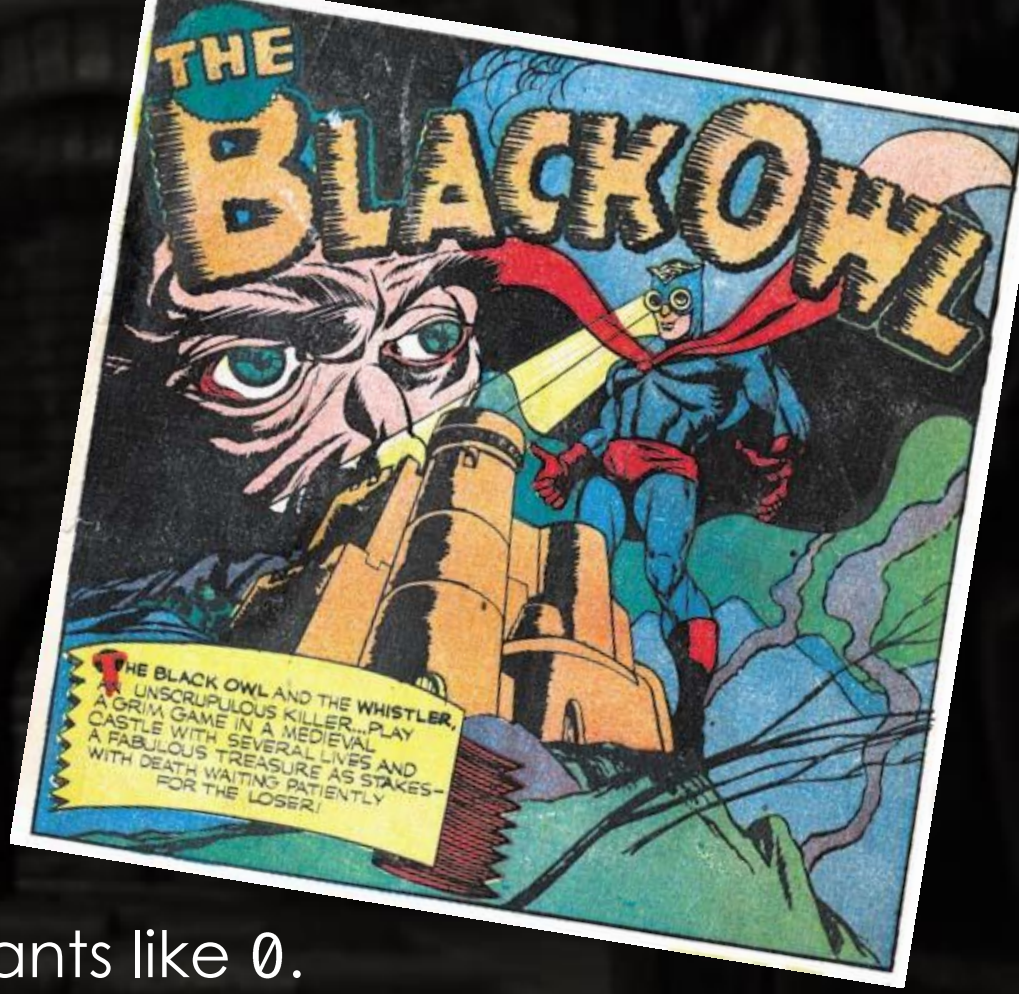
HT:@shafikyaghmour, @ubsanitizer

Owls

(0,0)

- An old MSVC warning suppression hack

```
while (0,0) { ... } // owly eyes

while (0)   { ... } // warning C4127: conditional expression is constant
```



- Used in MACROs
  - `#define FOO(X) do { f(X); } while (0,0)`
- From VS2015 suppress C4127 for trivial constants like `0`.

```
while (0)    { ... } // OK
while (42)   { ... } // warning C4127
```

"...no call to `operator[]()` may invalidate pointers, references or iterators..."
– Johnathan Wakely, SO 2015

Phantom Types

- Phantom Types make types Stronger!
- A template *type* parameter that is unused in the template implementation.
- Used as a *Tag* for type differentiation between otherwise identical types.

"Phantom types" is one technique that helps us to model the behaviour of our business logic in the type system. Illegal behaviour becomes a type error.

-- Ben Deane, CppCon 2016

Phantom Types

```cpp
template <typename T> // T is the "Phantom Type"
struct FormData
{
    explicit FormData(const string& input) : input_(input) {}
    std::string input_;
};

struct sanitized {};   // tag type
struct unsanitized {}; // tag type


// usage:
FormData<unsanitized> GetFormData();
std::optional<FormData<sanitized>> SanitizeFormData(const FormData<unsanitized>&);
void ExecuteQuery(const FormData<sanitized>&);
```

Using Types Effectively - Ben Deane - CppCon 2016

Poisoning Functions

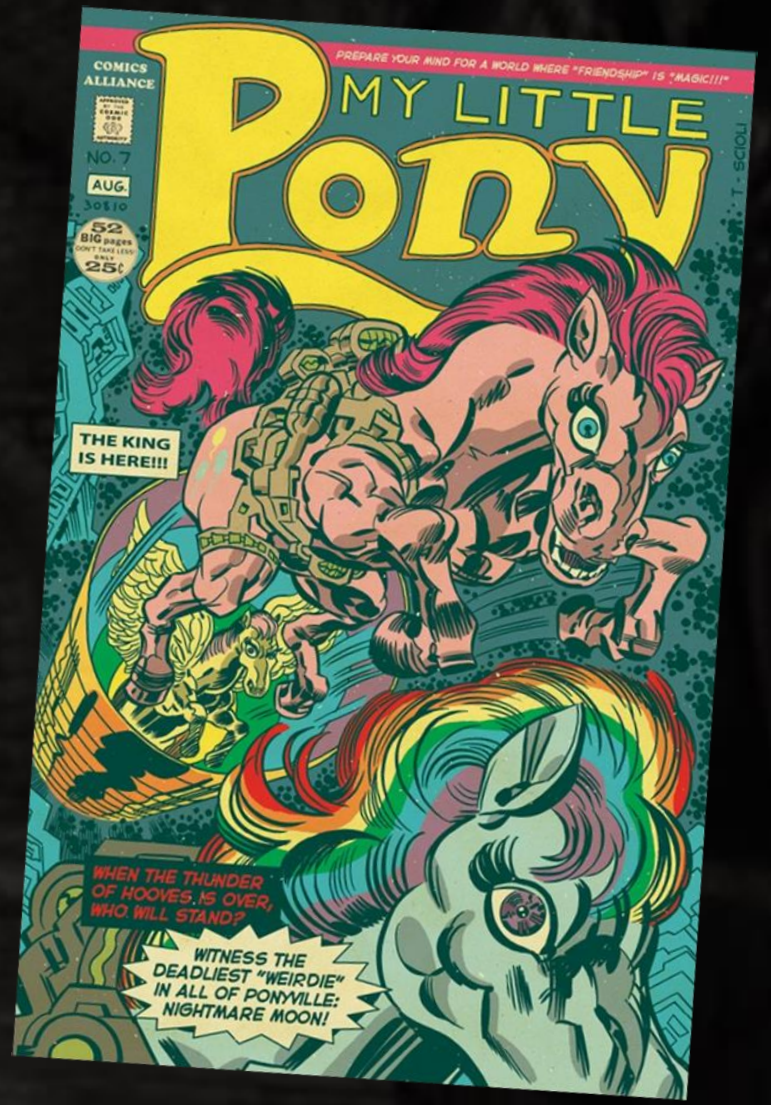
- A gcc/clang `#pragma` to "poison" an identifier
- Usage ➔ compile error
- Possible uses:
  - Identify deprecated features
  - Selectively prevent function usage from header (e.g. after #pragma)
- Non-standard, not portable
- Works on identifiers
  - No overload/namespace options

"..after leaving it for years collecting dust in the back of my mind, I ran into use cases where function poisoning allows to write more expressive and safer code"
-- Federico Kircheis, Fluent {C++}

Pony Princesses

- Some Atomic compare-and-exchange operations might fail when padding bits do not participate in the value representation of the active member.
- From The Holy C++20 Standard (Draft):
  The following code is not guaranteed to ever succeed:

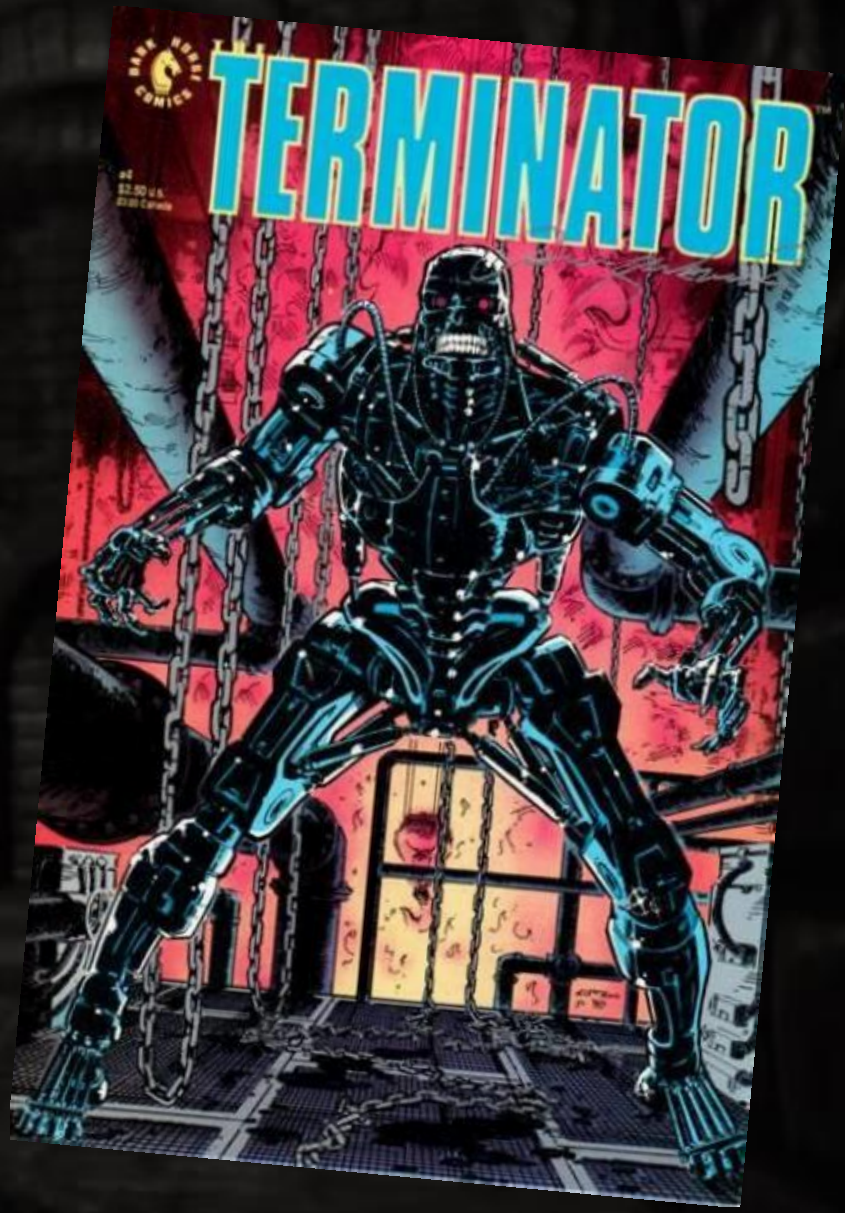
```
union pony {
  double celestia = 0.;
  short luna; // padded
};
atomic<pony> princesses = ATOMIC_VAR_INIT({});

bool party(pony desired) {
  pony expected;
  return princesses.compare_exchange_strong(expected, desired);
}
```



"...Best part was getting Very Serious Core People to say "the pony example" over and over as they debated wording and pointedly ignored the... quaint... example."
-- @jfbastien, Twitter

TERMINATOR

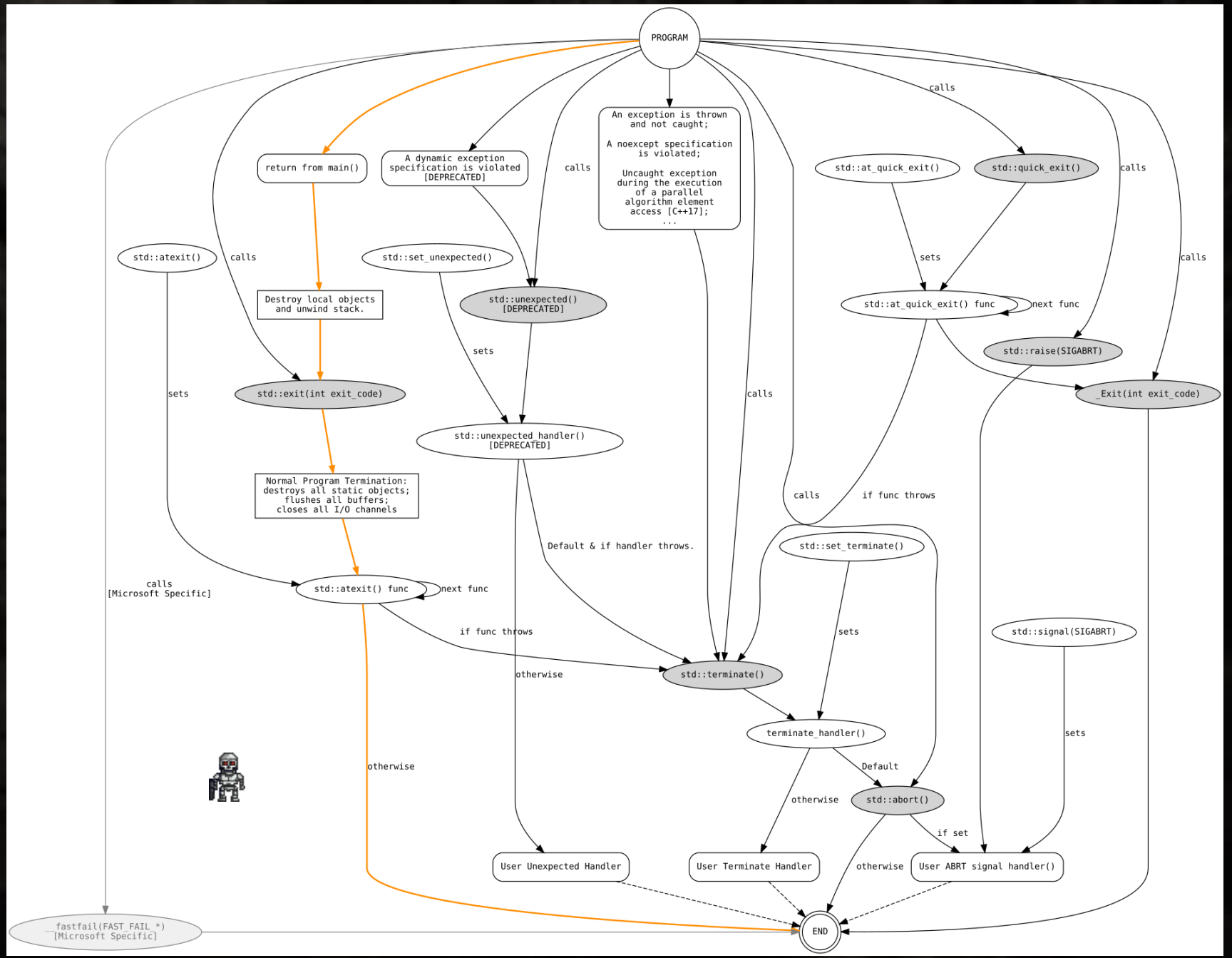
- Many ways to die in C++
- Important to know
- Also along module boundaries – DLLs
- Terminators include:
  - `std::exit()`
  - `std::abort()`
  - `std::terminate()`
  - `std::signal()`
  - `std::raise()`



- Complexly related to Destructors and `std::destroy()`

"Hasta la vista, baby!"
– Terminator

TERMINATORS



The Void

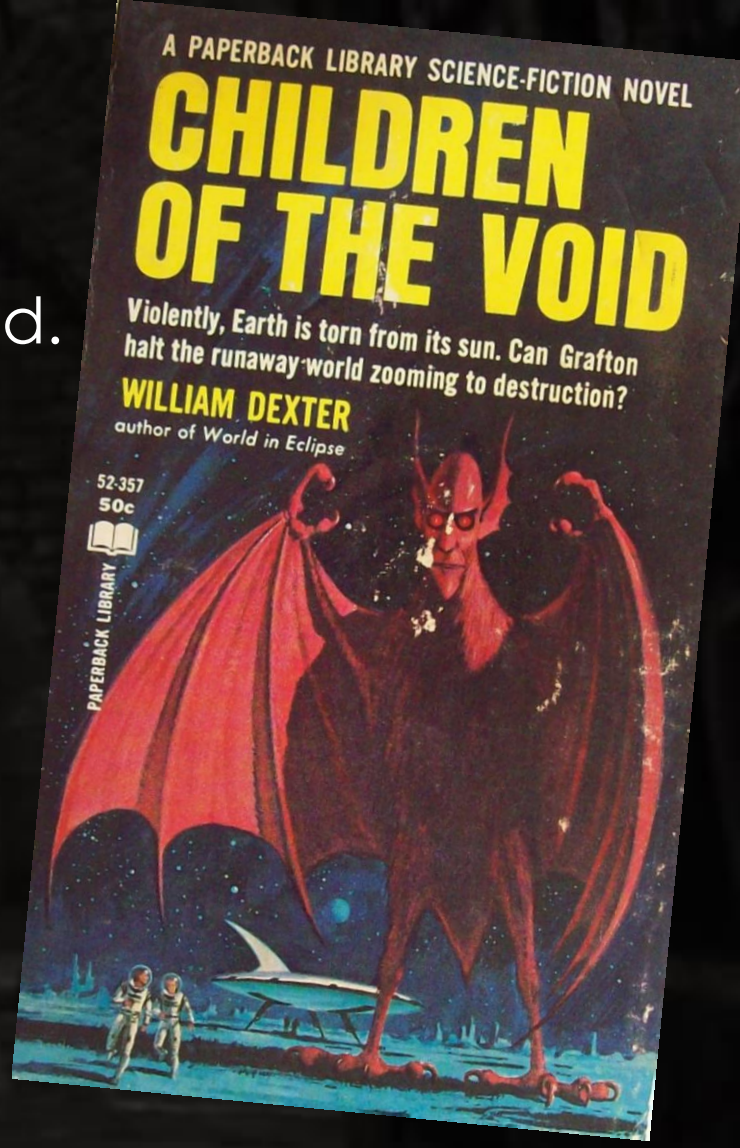
- `void`: an *incomplete* type that cannot be instantiated.
- But it is *not* the Empty Type.
- It is more like ~~the~~ a Unit Type
  - A type with only one value
  - Hence ⇒ no need to specify explicitly!
- Requires exceptional rules
  - [P0146R1] Regular Void
- An Empty Type (a.k.a `Void`)?

```
struct Void final
{
    Void() = delete;
};

auto absurd(Void);
```

BUT Wait!
There're More Unit Types!

`nullptr_t`
`nullopt_t`
* `monostate_t`
* `none_t`

** `std::void_t()`



"`void` is a bit weird—but it's not something a lot of people lose sleep over. Until it starts wreaking havoc on your generic code—because it's like the `vector<bool>` of the type system.
– Barry Revzin, Without Form and Void

Voldemort Types

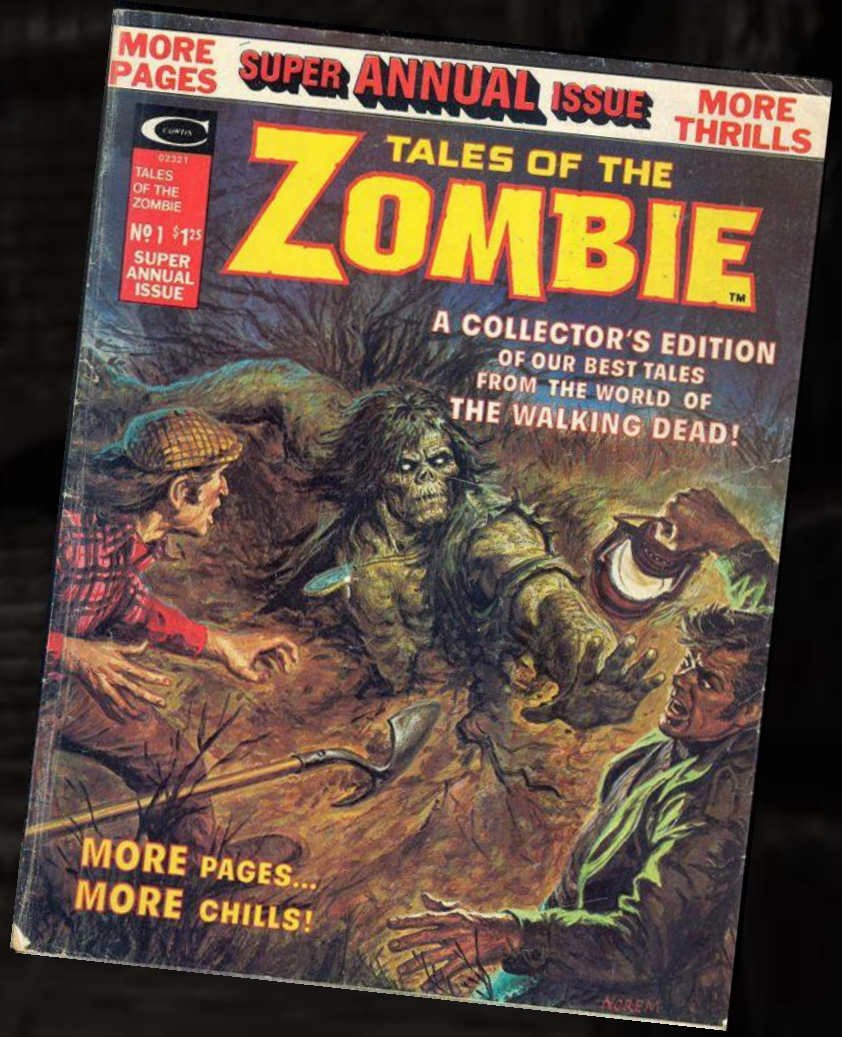
- A type that *cannot be named* outside of the scope it's declared in, but code outside the scope *can still use* this type.

```
int main()
{
    auto createVoldemortType = [] // use lambda auto return type
    {
        struct Voldemort          // localy defined type
        {
            int getValue() { return 21; }
        };
        return Voldemort{};       // return unnameable type
    };

  auto unnameable = createVoldemortType();  // must use auto!
  decltype(unnameable) unnameable2;         // but, can be used with decltype
  return unnameable.getValue() +            // can use unnameable API
         unnameable2.getValue();            // returns 42
}
```

"I can make things move without touching them."
– Lord Voldemort, A.K.A "He-Who-Must-Not-Be-Named", HP&HBP

Zombies

- What happens to an object in scope *after* it is moved-*from*?
- A zombie-like state:
  - Valid but unspecified?
  - Destructive move?
  - Only ops with NO pre-conditions

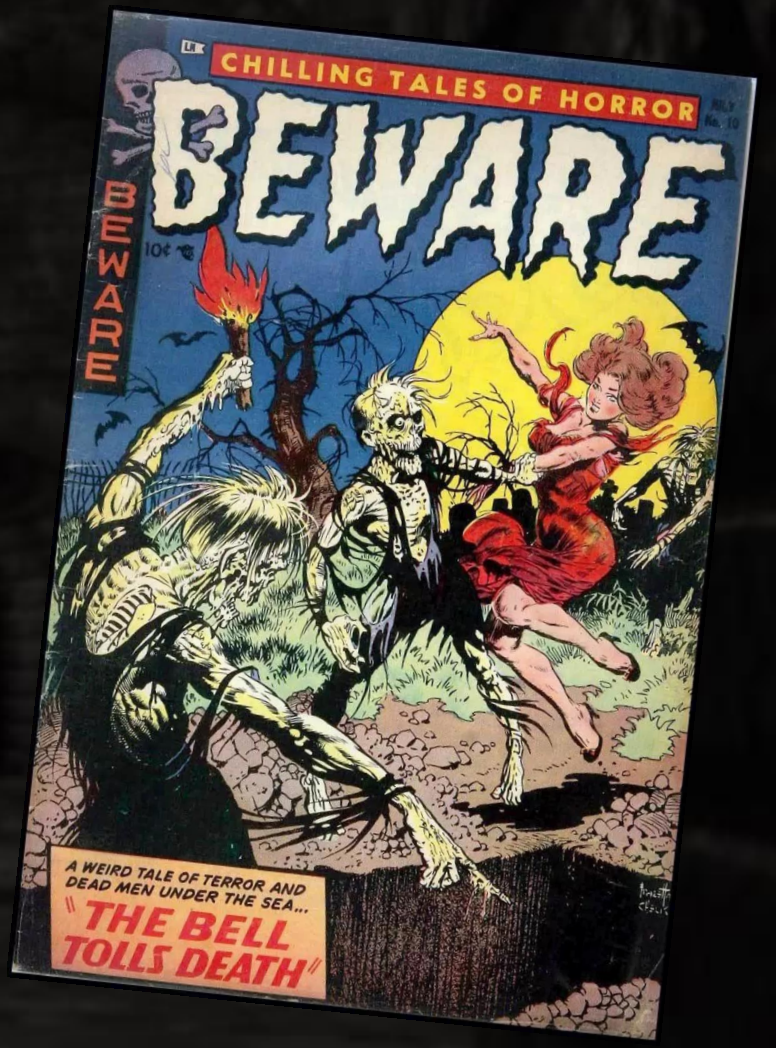


* Zombies are unrelated to `std::decay()`

"We've got zombies in the C++ standard. There are two factions: one of them stating that it is ok to have well defined zombies, while some people think that you'd better kill them."
– Jens Weller, C++ and Zombies

Zombies & Brains

- ISO C++ Standards: 20.5.4.3.1 *Zombie names*
- Index:
  - ***brains**, names that want to eat your, [zombie.names]*
  - ***living dead**, name of, [zombie.names]*
- The Standard's Crypt
- For standardized and later deprecated `std` names:
  - Including: `auto_ptr`, `binary_function`, `bind1st`, `bind2nd`, `random_shuffle`, `unary_function`, `unexpected` and `unexpected_handler`



"Brains: names that want to eat your, [zombie.names]"
– Richard Smith, [Stephan T. Lavavej], The Holy ISO C++ Standard Index

There are only two kinds of languages:
the ones people complain about and the ones nobody uses.

— Bjarne Stroustrup, The C++ Programming Language

videocortex.io/2017/Bestiary

Unicorns

- The *Unified Call Syntax* proposal
- `f(x,y)` can invoke a member function `x.f(y)` if there are no `f(x,y)`
- The inverse transformation: from `x.f(y)` → `f(x,y)`
  Not proposed
- Today needs 2 functions:
  - `begin(x)` and `x.begin()`
  - `swap(x,y)` and `x.swap(y)`
- Controversial → Might break Legacy code.
- *Not* in C++ [yet!]

"Good news everyone! I've implemented C++ Unicorn Call syntax"
— JF Bastien, Twitter 2016

Unicorn Call Syntax

```
struct 🦄 {
  🦄(int _🦄) : _🦄(_🦄) {}
  operator int() { return _🦄; }
  int _🦄;
};

🦄 operator ""_🦄(unsigned long long _🦄) { return _🦄; }

int main() {
  auto unicorn = 42_🦄;
  return unicorn;
}
```

Demons

- The notorious and infamous *Nasal Demon* of *Undefined Behavior (UB)*
- Ancient origins in C

- *UB* renders the *entire* program meaningless if certain rules of the language are violated.

- Compiler (actively) assumes UB *never* happens

"“Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose.”
– John F. Woods, comp.std.c 1992

UB Demons

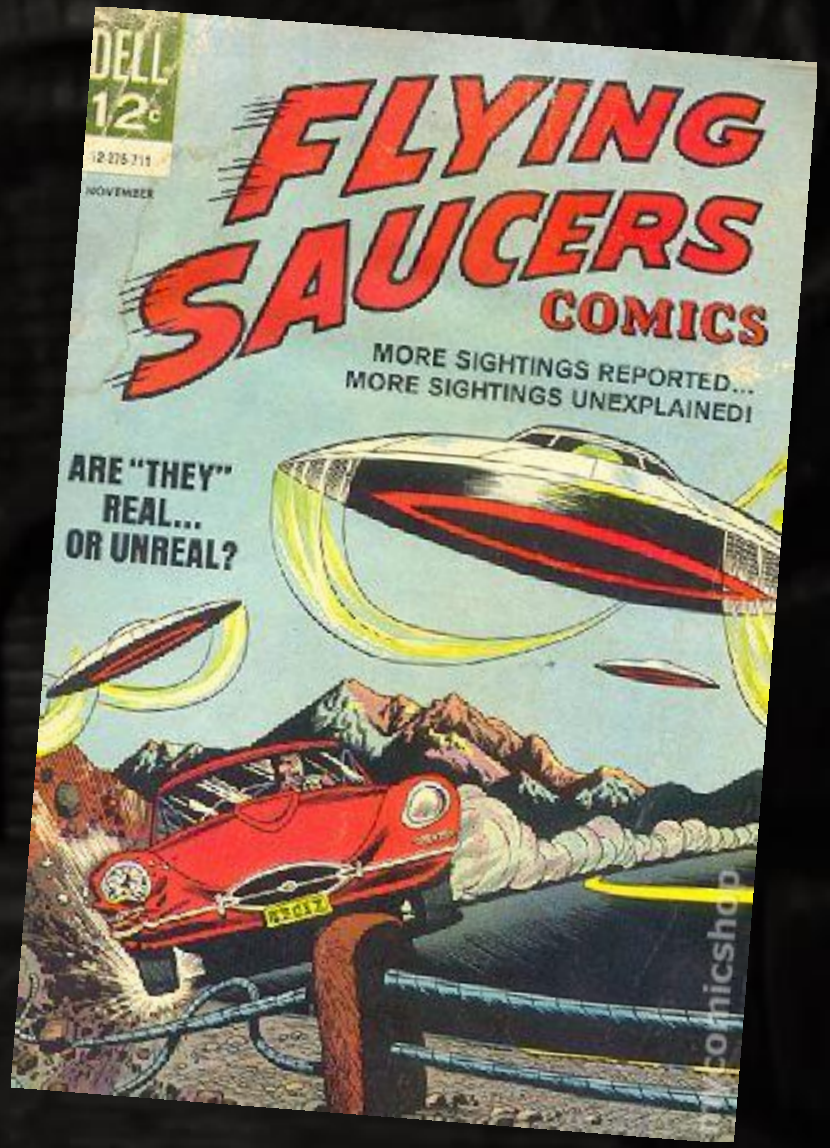
```
#include <cstdlib>                                  // for system()
typedef int (*Function)();                          // typedef function pointer type
static Function Do;                                 // define function pointer, default initialized to 0
static int EraseAll() { return system("rm -rf /"); } // naughty function
void NeverCalled()    { Do = EraseAll;             } // this function is never called!
int main()            { return Do();               } // call default-initialized function=UB: chaos ensues.
```

```
main:
        movl    $.L.str, %edi
        jmp     system

.L.str:
        .asciz  "rm -rf /"
```

Flying Saucers

- The <=> Spaceship Operator!
- Coming to C++20
- 3-way comparison
- Compiler automagically generates:
  < , <= , == , != , >= , >
- Consistent interface
- Support for partial ordering and more

- See [P0515R2]



"Unknown objects are operating under intelligent control...
It is imperative that we learn where UFOs come from and what their purpose is..."
– Admiral Hillenkoetter, First CIA Director 1960

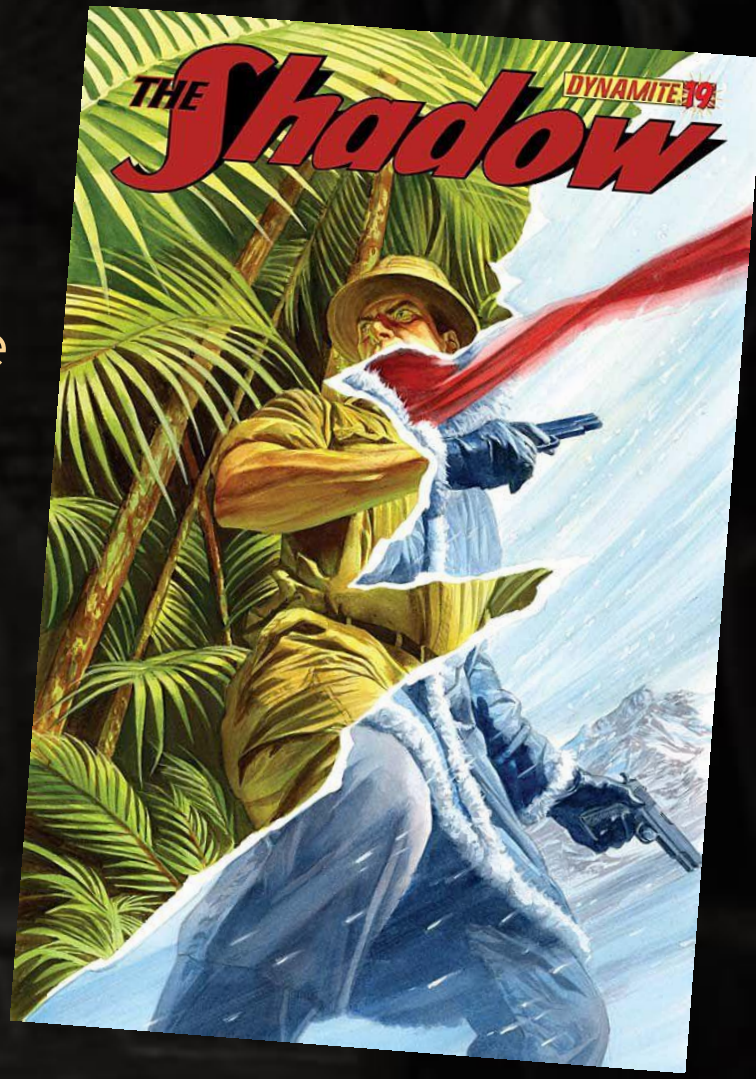
Flying Saucers


```
class ProverbialPoint
{
    int x, y;
public:
    auto operator<=>(Point const&) const = default; // totally-ordered member-wise comparison
                                                    ^^^^^^^^^^^^^^^^ ^^^^^^^^^^^
    // ...
};
```

Shadow Variables

- *Variable shadowing* occurs when a variable declared within a certain scope has the *same name* as a variable declared in an outer scope.
- Not limited to C++

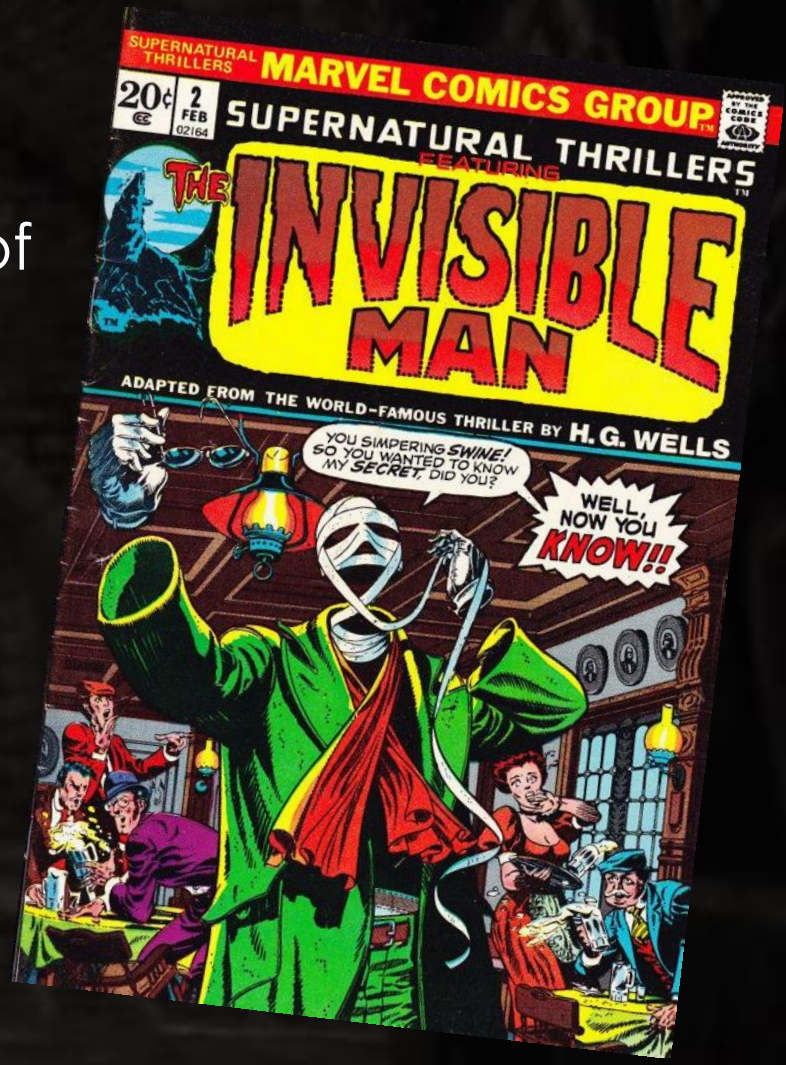
```
bool x = true;                                          // x is a bool
auto f(float x = 5.f) {                                 // x is a float
    for (int x = 0; x < 1; ++x) {                       // x is an int
        [x = std::string{"Boo!"}](){                    // x is a std::string
            { auto [x,_] = std::make_pair(42ul, nullptr);}  // x is now unsigned long
        }();
    }
}
```



"Only a lone foe could pierce that cordon; once inside, he would have to move by stealth, and strike with power and suddenness. I chose that mission."
-- The Shadow, Shadow Magazine #1311937

Transparent Objects

- A *transparent function object* accepts arguments of arbitrary types and uses *perfect forwarding*
- Avoids unnecessary copying and conversion when the function object is used in heterogeneous context, or with r-value arguments.
- Template functions such as `set::find()` and `set::lower_bound()` use on their Compare types
- Notable *Transparent Function Objects* include
  - `std::less<>`
  - `std::equal_to<>`.



"a thing and not a man; a child, or even std::less<> – a black amorphous thing."
— Ralph Ellison, Invisible Man