Extreme Type Safety with Opaque Typedefs

Kyle Markley

Motivating Example

- Microprocessors have many kinds of addresses
  - Virtual address
  - Linear address
  - Guest physical address
  - Host physical address
  - DDR address

- This invites confusion:

```
func(uint64_t address);
```

Common Mitigations

- Document with comment

```
func(uint64_t address);   // linear address
```

- Document with parameter name

```
func(uint64_t linear_address);
```

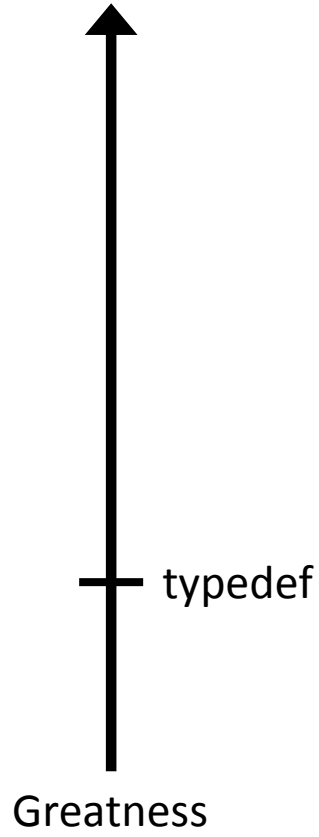
- Document with parameter type

```
using linaddr_t = uint64_t;
func(linaddr_t address);
```

Typedef – good, not great

- Semantically meaningful type names
  - Communicate your intent
  - Enable easy, comprehensive type changes
    - One line change to using/typedef declaration
    - cf. s/int/long int/ everywhere; risk breaking unrelated variables

- But typedef is *just an alias*, not a new type
  - func() call accepts any uint64_t, not only the intended linaddr_t
  - Cannot overload func() for different kinds of address
  - Unintended interoperability
    - e.g. linear_address + host_physical_address is merely uint64_t + uint64_t



Opaque Typedef

- A separate type, based on an existing type, but not an alias
- Proposed as a language feature
  - https://isocpp.org/files/papers/n3741.pdf

- You don't need to wait for that!

- Most of the value is possible with a library implementation (C++11)

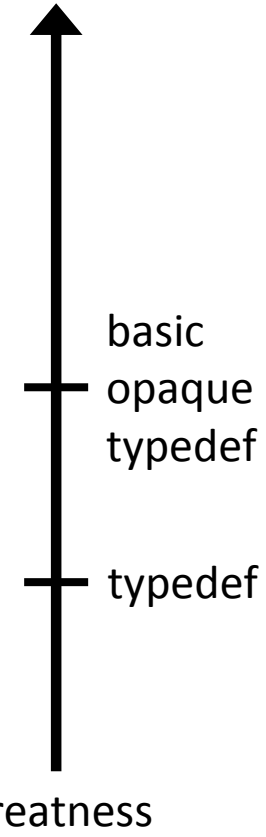
Basic Opaque Typedef

- Wrap a variable of some type in a new type
- Mimic the interface of the original type, but using the new type

```
// Just like a uint64_t, but no implicit conversions
struct linaddr : numeric_typedef<uint64_t, linaddr>
{
  using base = numeric_typedef<uint64_t, linaddr>;
  using base::base;  // explicit ctor takes uint64_t
}
```

That's pretty good!

- Just a few lines to create a numeric opaque typedef
- Safer interfaces by removing implicit convertibility
- Makes overloading on the new type possible



- Found a bug *the first time* I applied this technique
  - An assignment to an integer of a different width
  - Became a compile time error!

A foundation for more…

- You *don't* need to adhere to the interface of the original type
- Remove operations that are nonsensical – these are errors!
  - e.g. multiplying two addresses
- Add interoperability with other types
  - e.g. an "address" should interoperate with an "offset"
- Enforce semantics by controlling types of binary operators

address	+	address	→	error	address	-	address	→	offset
address	+	offset	→	address	address	-	offset	→	address
offset	+	address	→	address	offset	-	address	→	error
offset	+	offset	→	offset	offset	-	offset	→	offset

Specifying binary operators

- Obtain desired binary operators through inheritance
  - Library provides operator@ in terms of operator@=
  - User defines operator@= for other types
  - User specifies whether the operation is commutative
    - Given A::operator@=(const B&), enables “b @ a”
    - Don’t need to overload operator@= in *both* A and B
  - User specifies return type, parameter types, and optional parameter conversions

“I want commutative addition of address and offset to return an address”

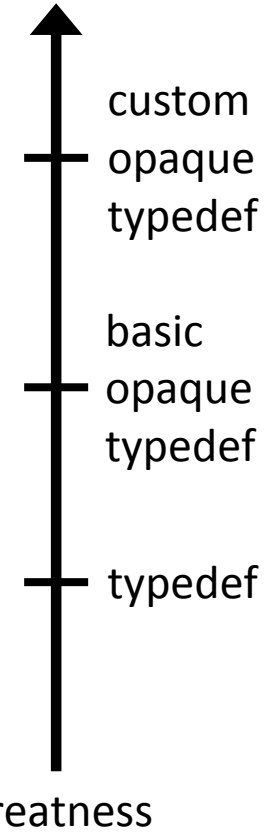
	op	return	commutative	param1	param2
Inherit from:	`addable<address, true, address, offset>`				
Inherit from:	`addable<address, true, offset, address>`				

Straightforward extensions

- Customize the default interface to add/delete operations
- If you want implicit conversions, you can provide them
- One line per binary operation involving other types

Hey, that's cool!

- Refining the interface
  - Turns semantic bugs into compile time errors
  - Enables richer, safer use of types
  - Often just one line per refinement
- No runtime penalty
  - Optimized away at -O1
- Technique is general
  - No need to limit to numeric types – how about safer strings?



I can haz code?

- Yes! Available under 3-clause BSD license
  - Header-only, about 1400 lines
  - gcc and clang like it; VS2015 does not, icc unknown

- UPDATED after the conference – the code is online, now:
  - http://sourceforge.net/projects/opaque-typedef/

- Submitted to the Boost Incubator

What I want in exchange ☺

- noexcept(auto)
  - "This function is transparent to exceptions. I don't throw any deliberately, here, but I am a function template and don't want to make any interface guarantees. I should be noexcept if and only if all the things I do when instantiated are also noexcept."

- Because this violates the Don't Repeat Yourself principle:

```
foo& operator+=(const foo& peer)
    noexcept(noexcept( value += peer.value )) {
                       value += peer.value;
    return *this;
}
```