Type Traits

What are they and why should I use them?

Marshall Clow
Qualcomm

mclow@qti.qualcomm.com

Twitter: @mclow
Occasional Blog: https://cplusplusmusings.wordpress.com/

What are type traits?

Type Traits are compile time template metafunctions that return information about types.

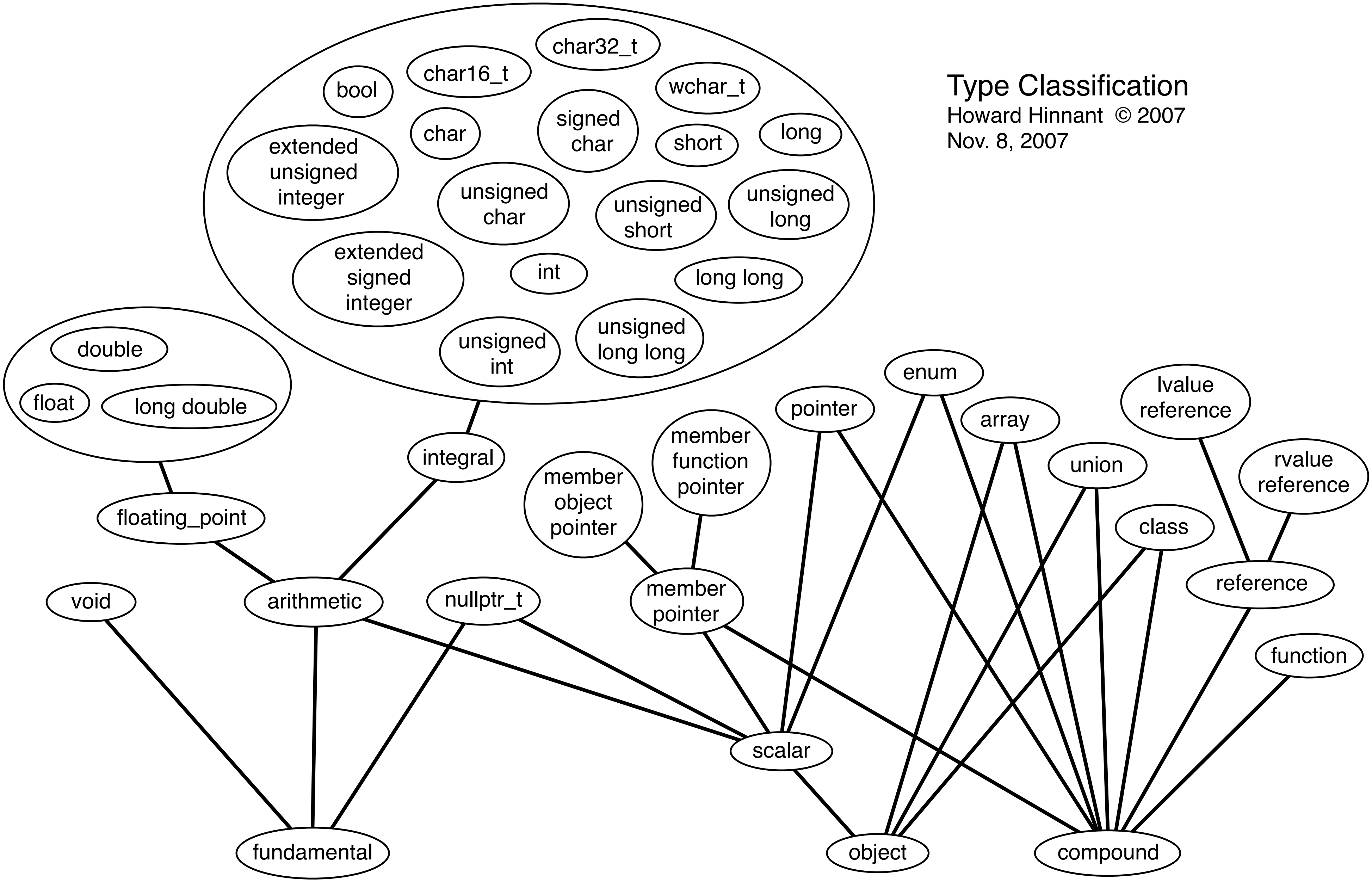
So what?

Sometimes, when writing generic code, you need to know “things” about the types of the information that you are manipulating.

How many different “kinds of types” are there?

How many different“kinds of types” are there?

- void
- integral
- floating point
- nullptr
- class
- array
- pointer
- rvalue reference
- lvalue reference
- union
- enum
- function
- member object pointer
- member function pointer



Type Classification

Howard Hinnant © 2007
Nov. 8, 2007

A type trait is a (templated) struct, and the member variable(s) and/or member types of the struct give you information about the type that it is templated on.

is_floating_point

- `std::is_floating_point<T>::value`
  - true for fp types (float, double, long double)
  - false for all other types

std::rank

```
* rank<int[5][2]>::value--> 2
* rank<int[5]   >::value--> 1
* rank<int      >::value--> 0
```

std::remove_const

- `remove_const<const int>::type` --> int
- `remove_const<      int>::type` --> int

There’s no reason that a type trait only return one result.

But why?

(1)
Writing Algorithms

* Iterator classification

* How can you manipulate the objects that you are dealing with

(2)
Restricting templates using enable_if

* You can restrict the availability of a templated class/function to types that have a particular property.

SFINAE Example

```
int func(...) { return 0; }

template <typename T>
typename std::enable_if
        <std::is_integral<T>::value, int>::type
func (T val) { return 1; }

int func(float f) { return 2;}

int main () {
   std::cout << func(nullptr) << " ";
   std::cout << func(2)       << " ";
   std::cout << func(2.f)     << " ";
   std::cout << func(2.0)     << std::endl;
   }
```

(3)
Provide optimized versions of generic code for some types

vector<T>::push_back

- Strong exception guarantee
- vector has capacity and size
- General case for push_back when **`size() == capacity()`**:
    - Allocate new memory, copy items to new memory, destruct old items, deallocate old memory

push::back (2)

- What if T is moveable?
    - Allocate new memory, move-construct the elements to the new memory, destruct old items, deallocate old memory
- Must be no throw move-constructible

push::back (3)

* What if T is trivially copyable?
    * Allocate new memory, memcpy bytes to new memory, destruct old items, deallocate old memory

push::back (4)

- These are all optimizations.
- None of them are required for correctness - but vector's users are happy that it does.

References

* Modern C++ Design by Andrei Alexandrescu

* C++ Template Metaprogramming by Dave Abrams & Aleksey Gurtovoy

Questions?

Thank you!