C++

April 11, 2022

Mike Spertus

spertus@uchicago.edu

MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

Type traits

Type traits

- Templates in the `<type_traits>` header giving useful properties of different types
- We use these all the time
  - Last lecture made essential use of `std::is_nothrow_move_constructible`
- They are a precursor to and basis for concepts
- They are also good for shaking off template rust as we move into the advanced study of templates
- And they provide a gentle introduction to "metafunctions"

Type traits as "metafunctions"

- You can think of type traits as “metafunctions”
  - Metafunctions typically take template arguments to generate code at compile time
    - Typically, a metafunction class template "returns" a type by putting a local type alias in the instantiated class named `type`
    - They can also return `int constexpr` values, addresses of functions, or any other compile-time constant expression etc. by putting a static member named `value` in the class
  - We will become metafunction experts later this quarter
- This is more confusing to describe than to use
- So let's see some type traits

Creating a trait

-
  ```
  template<class T> // False by default
  struct is_pointer {
    static bool constexpr value = false
  };

  // Partially-specialize to make true for pointers
  template <class T>
  struct is_pointer<T *> {
    static bool constexpr value = true;
  };

  static_assert(is_pointer<int *>::value); // int * is a pointer
  ```
- Normally, you'd use C++' type traits rather than building your own, but this shows how

`true_type` and `false_type`

- This is so common that the standard defines helper classes (roughly) like

-
  ```
  struct true_type {
    static bool constexpr value = true;
  };

  struct false_type {
    static bool constexpr value = false;
  };
  ```

An easier way to define it

-
  ```
  template<class T>
  struct is_pointer : public false_type {};

  template <class T>
  struct is_pointer<T *> : public true_type {};
  ```
- Not just easier, but better as it will let us select overloads in the optimized_copy example below
- Normally, you'd use C++11' type traits rather than building your own, but this shows how

A transforming trait

-
  ```
  template<class T>
  struct remove_pointer {
    using type = T; // Return transformed type in type
  };

  template <class T>
  struct remove_pointer<T *> {
    using type = T;
  };

  static_assert(is_same<int,
    remove_pointer<int *>::type>::value);
  static_assert(is_same<int,
    remove_pointer<int>::type>::value);
  ```

Making type traits easier to use

- Suppose I want to use `remove_pointer` in a template
- The following natural code doesn't work

-
  ```
  template<typename T>
  void f(T t) {
    remove_pointer<T>::type rt;
  }
  ```
- The intent that the type of `rt` is the "de-pointered" version of `T`

What went wrong

- Since the compiler doesn't know what T is
- It doesn't know if `remove_pointer<T>::type` is a type or a value
  - We're just using "type" as an English word. It doesn't mean anything to the compiler
- And misparses the class template ☹
- This is highly technical, but the simple solution is if the compiler is parsing a type as a value, you can help it along with the keyword typename
-
```
template<typename T>
void f(T t) {
  typename remove_pointer<T>::type rt;
}
```

To fix this and make code simple

- We have a type alias template

-
  ```
  template<typename T>
  using remove_pointer_t =
    typename remove_pointer<T>::type;
  ```
- Now our function is even easier
  ```
  template<typename T>
  void f(T t) {
    remove_pointer_t<T> rt;
  }
  ```
- So nice, we do it for values as well, even though there is no parsing ambiguity
- `static_assert(is_pointer_v<int *>);`

A trickier trait

- `is_convertible<A, B>` determines if A can be implicitly converted to B
- The trick is that we can do a lot of computation inside of the sizeof operator
  - sizeof needs to do full overload resolution on an expression and overload resolution is influenced by convertibility
- See is_convertible.h

An even harder trait

- `is_trivially_copy_assignable<T>`
- Unfortunately, no way to implement in a library function without compiler support?
- So the compiler generates an intrinsic

Do we really care about such crazy type traits?

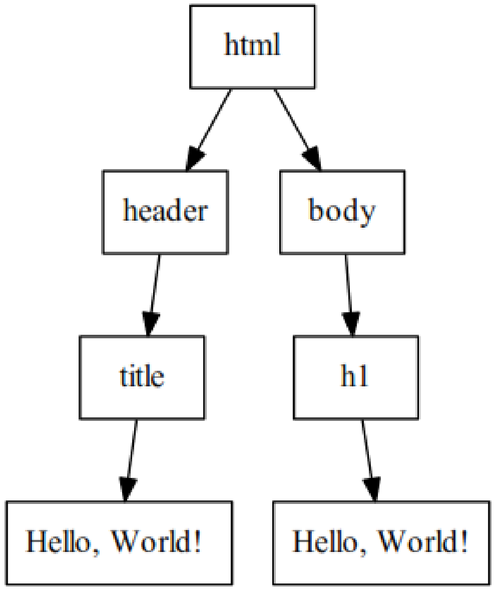
- Yes!
- Last week, we saw how `std::vector` couldn't even work without `is_nothrow_move_constructible`
- `is_trivially_copy_assignable` (finally) solves a problem we mentioned on week 1 of last quarter
- Let's turn back our time machine

Lightweight abstractions Example: Copying objects

- How do we copy one data structure to another?
- In C, we get low-level performance by copying the underlying memory with the `memcpy` command
- While this is fast, it’s not abstract, so it’s not fit for production programs that are complex and need to be evolved over time
- Let’s see why

“Compound” object

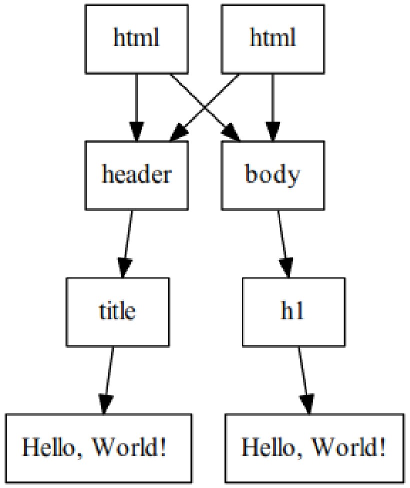
- More complex objects may (logically) contain other objects spread all over memory
- For example, an object representing an HTML page



Copying a compound object

- In C, if I copy a compound object, only the “root” is copied

-
  ```
  HTMLPage a;
  a = /* ... */; // "Hello" page
  b = a; // Oops! Only copies root
  ```
- One generally has to manually write “deep copying” commands in C, polluting the code with brittle implementation details
- Can we do better in C++?

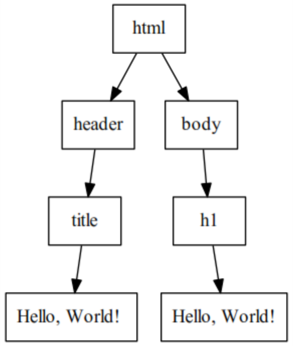


Copying compound obj in C++

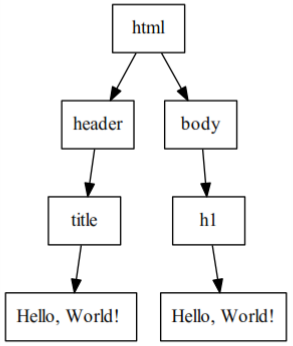
- We haven’t talked about classes yet, but for now, they are just the way you create your own types in C++
- C++ classes let you specify a “copy constructor” that explains the right way to copy objects belonging to that class

-
  ```
  class HTMLPage {
  public:
    // Copy constructor does deep copy
    HTMLPage(HTMLPage const &) { /* ... */ }
    /* ... */
  };

  HTMLPage a = /* ... */; // "Hello" page
  b = a; // Automatically does deep copy
  // Programmer doesn’t care if storage is contiguous
  ```
- **Abstract:** Just copy any object with an assignment, independently of implementation
- **Lightweight:** The compiler generates the code, so it is just as efficient as-if you did it manually
- b:



a:



Lightweight abstractions Example: Copying containers

- In Java, it is easy to create a new container that is a copy of an old container

-
  ```
  // https://javarevisited.blogspot.com/2014/03/how-to-clone-collection-in-java-deep-copy-vs-shallow.html
  Collection<Employee> copy = new HashSet<Employee>(org.size());

  Iterator<Employee> iterator = org.iterator();
  while(iterator.hasNext()){
      copy.add(iterator.next().clone());
  }
  ```
- This is very abstract, and doesn't care how the containers and objects are laid out in memory, so you can change from one type of container to another without breaking your code, etc.
- However, the price of this abstraction is poor performance as a contiguous array of primitive objects in memory is inefficiently copied one object at a time

copy: A lightweight abstraction

- As abstract as Java
  - Can copy any collection to any other collection with no change to code
    - Less risk of bugs, separates implementation from usage, reduces brittleness so programs can evolve
- As efficient as C
  - The compiler uses a mechanism called *template programming* to automatically generate the most efficient code at compile-time, so if a block-move memory would do the trick, the compiler will simply generate the same `memcpy` as C
    - This results in an 800% performance improvement in such cases. Since copying objects is very common, this is not an unimportant optimization
- We will learn how to write code that does this ~~over the course of the quarter~~ now!

`std::copy`

- C++ standard library provides a standard copy function that is the ultimate in abstraction
- It can copy from anything to anything else (anything from memory to containers to the console) and even does deep copies
- Example:
  - Copying from a `list<HTMLPage>` to `ostream_iterator<HTMLPage>(cout, ”\n”);` will copy each object individually
  - While opying from a vector<char> to a C array will do a `memcpy`

```
vector<char> v;
...
copy(v.begin(), v.end(), arr);
```

When can we use `memcpy` for assignment?

- That's what the type trait `is_trivially_copy_assignable` means!
- Let's look at an implementation
- optimized_copy.h

std::move

- Now, we can also understand how `move` is implemented
- Recall, `move` is a fancy way to convert any reference into an rvalue reference

-
  ```
  HTMLPage obfuscate(HTMLPage p);
  HTMLPage p1 = ...;
  serve(obfuscate(p1)); // Slow copy
  serve(obfuscate(static_cast<HTMLPage&&>(p1))); // Fast but ugly
  serve(obfuscate(move(p1))); // What we know and love
  ```
- Let's look at last quarter's implementation slide

How is std::move implemented? (Very advanced) Can do now

- Yet another “way of argument passing”
- If && is used in a template argument, it means “forwarding reference,” which bind to either an rvalue reference or a regular (lvalue) reference
- Template type deduction takes place according to the following “collapsing rules” which are only applied in templates
  - T& & ≅ T&
  - T& && ≅T&
  - T&& & ≅T&
  - T&& && ≅T&&

std::move code

-
  ```
  template <class T>
  remove_reference_t<T> &&
  move(T&& a)
  { return a; }
  ```
- What happens in the code
  ```
  A a;
  f(move(a));
  ```
- For what `T` is `T&&` an `A` or `A&`?
- By the collapsing rules, we see that the only option is that
  `T ≅ A&. (T && ≅ A& && ≅ A&)`
- Now, we return a
  `remove_reference_t<A&>&& ≅ A&&`
- Which tells `f` that `a` can be “raided for parts.”
- If you are interested, you can check that all the other cases work

Standard library type traits: Type categories

- is_void, is_null_pointer, is_integral, is_floating_point, is_array, is_enum, is_union, is_class, is_function, is_pointer, is_lvalue_reference, is_rvalue_reference, is_member_object_pointer, is_member_function_pointer, is_fundamental, is_arithmetic, is_scalar, is_object, is_compound, is_reference, is_member_pointer, is_const, is_volatile, is_trivial, is_trivially_copyable, is_standard_layout, is_literal_type, has_unique_object_representations, is_empty, is_polymorphic, is_abstract, is_final, is_aggregate, is_signed, is_unsigned, is_bounded_array, is_unbounded_array, is_scoped_enum

Standard library type traits: Operation properties

- `is_constructible`, `is_trivially_constructible`, `is_nothrow_constructible`
- `is_default_constructible`, `is_trivially_default_constructible`, `is_nothrow_default_constructible`
- `is_copy_constructible`, `is_trivially_copy_constructible`, `is_nothrow_copy_constructible`
- `is_move_constructible`, `is_trivially_move_constructible`, `is_nothrow_move_constructible`
- `is_assignable`, `is_trivially_assignable`, `is_nothrow_assignable`
- `is_copy_assignable`, `is_trivially_copy_assignable`, `is_nothrow_copy_assignable`
- `is_move_assignable`, `is_trivially_move_assignable`, `is_nothrow_move_assignable`
- `is_destructible`, `is_trivially_destructible`, `is_nothrow_destructible`
- `has_virtual_destructor`
- `is_swappable_with`, `is_swappable`, `is_nothrow_swappable_with`, `is_nothrow_swappable`

Standard library type traits: Properties and relation

- Properties
  - `alignment_of`, `rank`, `extent`
- Relations
  - `is_same`, `is_base_of`, `is_convertible`, `is_nothrow_convertible`, `is_layout_compatible`, `is_pointer_interconvertible_base_of`, `is_invokable`, `is_invokable_r`, `is_nothrow_invokable`, `is_nothrow_invokable_r`

Standard library type traits Transformation

- remove_cv, remove_const, remove_volatile, add_cv, add_const, add_volatile, remove_reference, add_lvalue_reference, add_rvalue_reference, remove_pointer, add_pointer, make_signed, make_unsigned, remove_extent, remove_all_extents, decay, remove_cvref, conditional, common_type, common_reference, basic_common_reference, underlying_type, invoke_result, type_identity

Standard library type traits: Misc

- Operations on traits
  - `conjunction, disjunction, negation`
- Member relationships
  - `is_pointer_interconvertible_with_class`
  - `is_corresponding_member`
- `is_constant_evaluated`

HW 12-1

- We discussed an implementation of `remove_pointer` that removes a pointer from a type
- `remove_pointer_t<int **>` is `int *`
- What if you wanted to remove all pointers?
- Define a `remove_all_pointers` trait that removes all pointers
- `remove_all_pointers_t<int **>` is `int`
- Don't forget to define its `_t` type alias template

HW 12-2

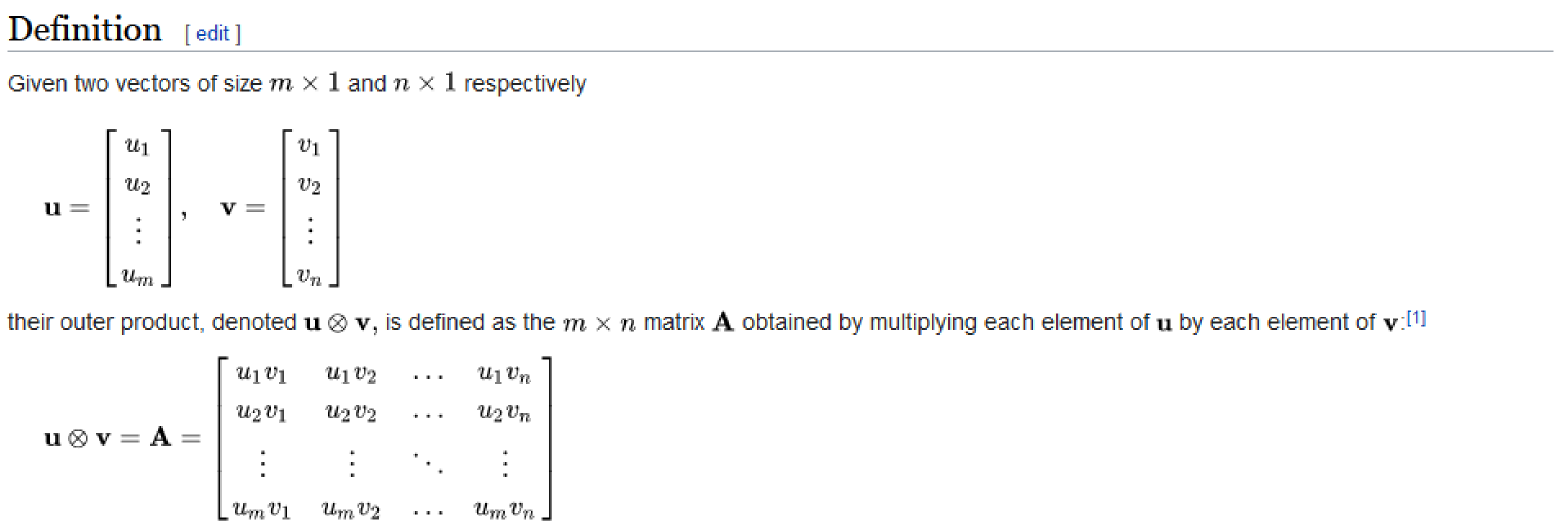
- Create your own implementations of the `is_reference` and `remove_reference` type traits
- Make sure they handle both rvalue references and lvalue references
- Be sure to define any relevant `_t` and `_v` versions
- Documentation like the cppreference page for `is_reference` often includes sample implementations, so you'll need to be careful not to inadvertently look at

HW 12-3

- Let's write a simple type printer
- `TypePrinter<int>()` will return the string `"int"`
- `TypePrinter<int const *>()` will return `"int const *"`
- `TypePrinter<int * const>()` will return `"int * const"`
- For simplicity and because of limitations to existing type traits, assume that the only constituents of the types you are printing are
  - `char, int, long, float, double, *, &, &&, const, volatile`

HW 12-4: Extra credit

- The "outer product" of two vectors shows all possible products
- According to https://en.wikipedia.org/wiki/Outer_product



Definition [edit]

Given two vectors of size $m \times 1$ and $n \times 1$ respectively

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

their outer product, denoted $\mathbf{u} \otimes \mathbf{v}$, is defined as the $m \times n$ matrix $\mathbf{A}$ obtained by multiplying each element of $\mathbf{u}$ by each element of $\mathbf{v}$:[1]

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{A} = \begin{bmatrix} u_1v_1 & u_1v_2 & \dots & u_1v_n \\ u_2v_1 & u_2v_2 & \dots & u_2v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_mv_1 & u_mv_2 & \dots & u_mv_n \end{bmatrix}$$

HW 12-4: Continued

- We want to write an `outer_product` function template that takes two vectors (possibly of different types) and returns a vector of vectors

-
  ```
  vector<int> vi = { 1, 2, 3 };
  vector<double> vd = { 4.1, 5.1};
  ```
- Then `outer_product(vi, vd)` will return the following `vector<vector<double>>`

-
  ```
  {{4.1, 5.1},
   {8.2, 5.2},
   {12.3, 5.3}}
  ```

HW 12-4: Continued

- The tricky part is going to be coming up with the return type of `outer_product`

-
  ```
  template<typename U, typename V>
  vector<vector<???>>
  outer_product(vector<U> u, vector<V> v);
  ```
- `???` needs to be replaced by the type you get when you multiple a `U` and a `V`
- This is especially tricky because operator overloading means that any two types might have a product

HW 12-4: Continued

- Begin by creating a `product_t<U, V>` type trait that gives the type that results from multiplying a `U` object by a `V` object
- Do *not* assume that `U` and `V` are default constructible
- Hint: The `declval` tricks from `is_convertible.h` will come in handy here
- You will get half credit for getting this far

HW 12-4: Continued

- Now implement `outer_product`
- Again, be careful not to assume that the product type is default constructible