C++

April 11, 2022

Mike Spertus

spertus@uchicago.edu

MASTERS PROGRAM IN COMPUTER SCIENCE

THE UNIVERSITY OF CHICAGO

Projects

- Consider starting to think about a project choice
  - No need to start coding yet, but you are encouraged to come up with an idea in the next 1-2 weeks
  - Feel free to consult with me if you aren't sure what will be suitable
- No requirements other than demonstrating your facility with a number of the techniques taught during this course
  - It doesn't need to cover all of them
    - E.g., if you have a lot of advanced template code that could compensate for not having advanced concurrency

ADVANCED VARIADICS

Variadics

- A variadic template is a template that can take a variable number of parameters called a parameter pack
- Here's how to create a function that adds up all of its arguments

```
template<typename T>
T adder(T v) {
  return v;
}

template<typename T, typename... Args>
std::common_type_t<T, Args...> adder(T first, Args... args)
{
  return first + adder(args...);
}
```

  - For example, `adder(1, 2, 3, 4)` will be `10`, and `adder("foo"s, "bar"s)` returns the string `"foobar"s`
    - Reminder: The `s` after the closing double-quote is a string literal

Pack Expansion

- ... is used to expand parameter packs into comma-separated lists (roughly)
- In the previous slide, we expanded `args...` to 2, 3, 4

What can be expanded by ...?

- ... doesn't have to be applied directly to a pack
- It can be applied to any expression that contains a pack
- (args + 2)... expands to 3, 4, 5
- https://godbolt.org/z/dfdfjfKqW

Expanding an expression with variadic parameter packs

- Everything inside the expression gets expanded and separated by commas
- ```
  // A version that takes its arguments by const reference
  template<typename T>
  T adder_ref(T const &v) {
    return v;
  }
  // Args const &... expands each argument,
  // puts a const & after it and separates
  // it with commas
  template<typename T, typename... Args>
  common_type_t<T, Args...>
  adder_ref(T const &first, Args const &... args) {
    return first + adder(args...);
  }
  ```
  ```

What if there are two packs in an expansion

- If multiple packs appear in an expansion, they are expanded in "parallel"
- They must have the same length
- (args*args)... expands to 1, 4, 9

Wait, we've seen that before!

- This "simultaneously expanding internal packs in expressions" is what makes perfect forwarding work!
- Consider the implementation of `make_unique` from last quarter
- ```
  template<typename T, typename ...Args>
  unique_ptr<T> make_unique(Args&& ...args) {
    return new T(forward<Args>(args)...);
  }
  ```
  ```

Variadics case study: Improved printing

- Nearly every program does formatted output
- Yet the two built-in ways provided by C++, IOStreams and printf have serious problems
- We will review the problems described in the prelecture
- and then show how to fix them with variadics

The problem with IOStreams

- C++ replaced "printf"-based I/O with I/O streams like you learned
- The problem is that IOStreams are very cumbersome for many simple tasks
- For example, we often see code like:
  - ```
    } catch(MyException e) {
      ostringstream msg;
      msg << "Operation "
          << e.operation
          << " failed with error code "
          << e.errorCode;
      Logger.output(msg.str()); // Some logging framework
    }
    ```
  - In fact, what you more often see in this case, is a lame "Error detected" message to avoid the 6 lines above
  - We'd rather write
    `printf("Operation %s failed with error code %d", e.operation, e.error_code);`
    ```

The problems with `printf`

- Unsurprisingly, many people continue to use `printf` with C++
- However, `printf` is not extensible
  - It doesn’t know how to print any new types you have created
- Even worse, it ignores argument types and prototypes and crashes and corrupts the stack at runtime

This code compiles without warning but crashes

- ```
  int main()
  {
    printf("%s\n", 1, 2);
  }
  ```
- The problem is that unlike nearly every other function, printf doesn’t declare or check the type or number of arguments it was called with because that depends on the format string
- `int printf(char *fmt, ...);`
- The `...` could be anything
  - Don't confuse with C++ variadics
  ```

Solution: C++20 format

- We can use variadic templates to create a typesafe extensible formatting library that is easier to use than `printf`
- `cout << format("id {} - name {}", 5, "Mike");`
- Automatically uses the right formatter
  - So I don't need to specify the type in the format string like I did with `printf`

Implement with variadics

- Let's practice with variadics by implementing a simplified version of `format` ourselves
- https://godbolt.org/z/bnn7ss33s

First handle the base case

- A typical approach in variadics is to have a non-variadic "base case" overload and then recurse
- This case is easy for `format`
  - No arguments to format!
- ```
  string simple_format(string_view fmt) {
    return string(fmt);
  }
  ```
  ```

How to print an argument

- As we saw in the prelecture, formatters for format are really complicated
  - And not really relevant for this variadics discussion
- We'll just use the stream inserter to print a type
- ```
  template<typename T>
  string make_string(T const &t) {
      ostringstream oss;
      oss << t;
      return oss.str();
  }
  ```
- IRL this would be to slow and clunky, but…
  ```

Now recurse and we're done

```
template<typename T, typename ...Ts>
string simple_format(string_view fmt, T const&t, Ts const &... ts)
{
    auto braces = fmt.find("{}"); // Omitting error handling :(
    return string(fmt.substr(0, braces)) + make_string(t)
           + simple_format(fmt.substr(braces + 2), ts...);
}
```

Fold operations

- This kind of recursing down a variadic parameter pack is very common
- But it can be verbose and clunky for simple things
- Look at our `adder` implementation
- C++17 added fold operations to simplify
- You can think of fold operations as a variadic analog to `accumulate`

Fold operations

- Putting the `...` before a binary operator followed by something containing a pack puts the operator between every element in the pack
- Sounds complicated but really isn't
- For example, if `args...` is `1`, `2`, `3`, then
  ```
  ... + args
  ```
  evaluates to
  ```
  1+2+3
  ```

adder redux

- Now, we can write `adder` simply without many levels of recursion
-
```
template<typename... Args>
common_type_t<Args...>
adder(Args... args) {
  return (... + args);
}
```

There are some variations on fold expressions

- In addition to the version shown here
- There are variants of fold expressions that
  - Start with an initial "accumulator" value
  - Fold from the right rather than the left
- They otherwise work the same as what we did and should be straightforward to pick up if you need them

Tuples

- We’ve been using tuples a lot
- Now we have the tools to understand them more deeply
- Tuples are a generalization of `std::pair` to any number of fields and the most important and prevalent example of variadics
- ```
  tuple<string, int, double> si("str", 2, 3.5);
  cout << get<0>(si) // prints "str"
  cout << get<int>(si); // prints 2 (C++14)
  int two = get<1>(si);
  ```
  ```

Why isn't `get` a member function?

- In the previous slide, why do we say `get<1>(di)` instead of the more natural
`di.get<1>()`?
- Consider the following:
```
template<typename ...T>
auto f(tuple<T...> t)
{ return t.get<0>(); }
```
- The compiler might think that `t.get` is a field in `t` and interpret it as asking whether `t.get` is less than `0`!
- In order to disambiguate, you would have to do the ugly
```
template<typename ...T>
auto f(tuple<T...> t)
{ return t.template get<0>(); }
```
- Kind of reminiscent of the ugly `typename` disambiguator mentioned above
- Since we didn't want such an important basic vocabulary class to require using such an advanced technique, we made `get` into a global function

Let's try to implement tuples

- Tuple.h

Returning multiple values

- We already learned about using tuples to return multiple values at runtime
- `pair<int, int> f() { return {1, 2}; // ok }`
- `tuple<int, int, char> f() { return {1, 2, ’u’}; } // OK in C++17`
- `auto [i, j, c] = f(); // Structured binding back into variables`

A purely compile-time use of tuples

- Often, one uses tuples as a sort of compile-time “typelist”
  - These tuples are not necessarily meant to be instantiated at run-time
  - Instead, we are using the tuple type as a convenience container of types at compile-time
- The next few slides will show how to manipulate typelists
- We will use these techniques in our tuple implementation
- For illustration, assume that the following types have been defined.
  - Note that we use tuple just a “compile-time container of types” to make it easier to organize variadic template arguments

```
struct A {
  static void foo() { cout << "In A" << endl; }
};
struct B {
  static void foo() { cout << "In B" << endl; }
};
using AB = tuple<A, B>;
Using CharTypes = tuple<char,unsigned char, signed char>;
```

Calculating length

```
template<class TList> struct Length;
template<>
struct Length<tuple<>>
{
  static size_t constexpr value = 0;
};
template<class T, typename... Us>
struct Length<tuple<T, Us...> >
{
  // Of course, we could just use sizeof...(Us),
  // but let's see how to do ourselves
  static size_t constexpr value
     = 1 + Length<tuple<Us...>>::value };
};
int main() // Prints 3
{
  cout << Length<CharTypes>::value << endl;
}
```

Finding the type at a given index

```
template<class List, int i> struct TypeAt;
template<class Head, typename... Tail>
struct TypeAt<tuple<Head, Tail...>, 0>
{
    using type = Head;
};
template<class Head, typename... Tail, int i>
struct TypeAt<tuple<Head, Tail...>, i>
  : public TypeAt<tuple<Tail...>, i - 1> // Inherits this result
{
};

int main() // prints "In B"
{
    // Note that I have to say TypeAt<xxx>::type to get the result
   TypeAt<AB,1>::type::foo();
}
```

Finding the index of a given type

```
template<class List, class Target> struct IndexOf;
template<class Target>
struct IndexOf<tuple<>, Target>
{
    static size_t constexpr value = -1; // Return -1 if not found
};
template<class ...Tail, typename Target>
struct IndexOf<tuple<Target, Tail...>, Target>
{
    static size_t constexpr value = 0;
};
template<class Head, typename... Tail, class Target>
struct IndexOf<tuple<Head, Tail...>, Target>
{
private: // Using a Compile-time temporary
    static size_t constexpr temp = IndexOf<tuple<Tail...>, Target>::value;
public:
    static size_t constexpr value = temp == -1 ? -1 : 1 + temp;
};

int main() // Prints 1
{
  cout << IndexOf<CharTypes, unsigned char>::value;
}
```

Appending the types in two tuples

```
template<class First, class Second> struct Append;

template<typename... Ts, typename... Us>
struct Append<tuple<Ts...>, tuple<Us...> >
{
    using type = tuple<Ts..., Us...>;
};

int main() // Prints 2
{
  typedef  Append<AB, tuple<int>>::type ABint;
  cout << IndexOf<ABint, int>::value;
}
```

Replacing a type in a tuple

```
template<typename T, typename A, typename B> struct Replace;

template<typename... Ts, typename A, typename B>
struct replace<tuple<A, Ts...>, A, B> {
  using type = tuple<B, Ts...>;
};

template<typename H, typename... Ts, typename A, typename B>
struct Replace<tuple<H, Ts...>, A, B>
 : public Append<tuple<H>, typename Replace<tuple<Ts...>, A, B>::type> {
};
```

- Why did I have to say `typename` in front of `Replace<Replace<tuple<Ts...>, A, B>::type`?
- The point is that when the compiler is looking at the above specialization, it doesn't know what types will be in `Ts…`, so it can't assume that `Replace<tuple<Ts…>, A, B>::type` is a type or not. (For example, it might be a static member variable). We clarify this to the compiler by preceding it with the keyword `typename`.
- If you are getting compiler errors because the compiler can't figure out that something is a type, just preceded it with the term "`typename`"

The Empty Base Optimization

- In C++, the address of an object is also the identity of the object
- For example, if a1 and a2 are two different a objects, then they should have different addresses
- This could break if they have size 0 (e.g., an array of size 0 objects)
- So even an empty class has size > 0!
- However, an empty base class does not take space since the total object addresses
  - Food for thought: Does it always address?

Simulating template virtuals

- We will put together a lot of template techniques for a really mind-bending payoff
- We will need one more tool at our disposal
- Just like regular methods, we sometimes with that method templates were virtual
- However, method templates cannot be virtual (Why?)
- Let’s see if we can come up with a way to simulate virtual method templates

What would template virtuals look like?

- The reason method templates can’t be virtual is because they aren’t methods
  - What would the vtable even look like?
- However, we can call ordinary methods from our method templates
- So what we will do is specify which specializations should call which method

What interface do we want to support

- Suppose we have a method template
-
```
struct S {
  template<typename T> void f();
};
```
- Now suppose we want `f<int>` and `f<double>`, and `f<A>` to be “virtuals” that there is a way to override them in a derived class (A is just another class)

First swing

- A first attempt might be something like:

```
struct S {
  template<class T>
  void f() { return fHelper(T()); }
  virtual void fHelper(int &&) = 0;
  virtual void fHelper(double &&) = 0;
  virtual void fHelper(A &&) = 0;
};
```

- This actually works for the int and double case
- A class that inherits from S can override the fHelpers
- Unfortunately, A may be abstract or not have a default constructor, so fHelper(`A()`) won't be able to create an `A`

TT

- What we need is a “tag type” that can always be created
  - Not abstract
  - Default constructible
  - Cheap to construct
-
```
template<class T>
struct TT {
};
```
- Fits the bill. If I construct with `TT<A>{}`, it’s legal because it is not abstract and is default constructible, and it’s cheap because it’s just an empty class

Better

- Now, improve our previous approach:

```
struct S {
  template<class T>
  void f() { return fHelper(TT<T>()); }
  virtual void fHelper(TT<int> &&) = 0;
  virtual void fHelper(TT<double> &&) = 0;
  virtual void fHelper(TT<A> &&) = 0;
};
```

- A class that inherits from S can override the fHelper()s

Can we automate?

- The only problem is that it is tedious and error prone to manually define all of the helper virtual methods
- Sounds like a job for variadics
- Let’s begin by defining a class template that holds just one Helper
- And then use variadics to inherit from all of them

The solution

```
template<typename T>
struct Holder {
  virtual void fHelper(TT<T> &&) =0;
};
template<typename ...Ts> // which holders to inherit
struct ST : public Holder<Ts>... {
  template<class T>
  void f() { return fHelper(TT<T>()); }
};

using S = ST<int, double, A>;
```

- S has virtual fHelpers for int, double, and A

Wow, that seems obscure
Why do I care?

- Well, you don't yet
- Since our mindbending application will need this
- For now, just suspend your disbelief

HW 13-1

- Create a `ReplaceAll<TL, T, U>` template that replaces all occurrence of `T` in `TL` with `U`.
  - For example, `ReplaceAll<tuple<char, int, char>,char, double>::type` should be `tuple<double, int, double>`
- Define a `Reverse` template that reverses the parameters of a tuple
  - For example, `Reverse<tuple<A, B>>::type` will be `tuple<B, A>`

HW 13-2

- Extend Tuple2 (on canvas) so that you can get elements by type as well as index
  - ```
    Tuple2<int, double> t2id(4, 5.6);
    cout << "get<1>(t2id) =  " << get<1>(t2id) << endl; // Works now
    cout << "get<int>(t2id) = " << get<int>(t2id) << endl; // HW
    ```
    ```

HW 13-3: Extra Credit

- The "outer product" HW for last week took the outer product of vectors
- Why wouldn't the approach work for arrays?
  - E.g., std::array<A, 5>
- How would you make it work?
  - **Warning:** Surprisingly difficult