How can I give a C++ lambda expression more than one operator()?

devblogs.microsoft.com/oldnewthing/20191107-00

November 7, 2019

Raymond Chen

Suppose you have stored a C++ lambda expression into a variable, and you want to call it in different ways. This seems impossible, because when you define the lambda expression, you can provide only one `operator()`:

```
auto lambda = [captures](int v) { return v + 2; };
```

This lambda has only one way of calling it: You pass an integer and it returns an integer.

But it turns out that you can create a lambda that can be called in multiple ways: Use an `auto` parameter!

```
auto lambda = [](auto p)
{
  if constexpr (std::is_same_v<decltype(p), int>) {
   return p + 1;
  } else {
   return "oops";
  }
};

auto result1 = lambda(123); // result1 is 124
auto result2 = lambda('x'); // result2 is "oops"
```

By declaring the parameter as `auto`, the lambda accepts any single parameter. We then use `if constexpr` and `std::is_same_v` to see what type was actually passed, and implement the desired function body for each type.

Notice that the different branches of the `if` don't need to agree on the return type. In our example, passing an integer adds one and produces another integer. But passing anything else returns the string `"oops"`!

You can create a bunch of tag types to make it look almost as if your lambda had member functions.

```
struct add_tax_t {};
constexpr add_tax_t add_tax;

struct apply_discount_t {};
constexpr apply_discount_t apply_discount;

auto lambda = [total](auto op, auto value) mutable
{
  using Op = decltype(op);
  if constexpr (std::is_same_v<Op, add_tax_t>) {
   total += total * value; // value is the tax rate
   return total;
  } else if constexpr (std::is_same_v<Op, apply_discount_t>) {
   total -= std::max(value, total); // value is the discount
   return total;
  } else {
   static_assert(!sizeof(Op*), "Don't know what you are asking me to do.");
  }
};

lambda(apply_discount, 5.00); // apply $5 discount
lambda(add_tax, 0.10); // add 10% tax
```

So far, all of our “methods” have the same number of parameters, but you can use a parameter pack to permit different numbers of parameters:

```
auto lambda = [total](auto op, auto... args) mutable
{
  using Op = decltype(op);
  using ArgsT = std::tuple<decltype(args)...>;
  if constexpr (std::is_same_v<Op, add_tax_t>) {
   auto [tax_rate] = ArgsT(args...);
   total += total * tax_rate;
   return total;
  } else if constexpr (std::is_same_v<Op, apply_discount_t>) {
   auto [amount, expiration] = ArgsT(args...);
   if (expiration < now()) {
     total -= std::max(amount, total);
   }
   return total;
  } else {
   static_assert(!sizeof(Op*), "Don't know what you are asking me to do.");
  }
};
```

In this case, the `add_tax` “method” takes a single parameter, whereas the `apply_discount` “method” takes two.

You could even dispatch based solely on the types and arity.

```
auto lambda = [total](auto... args) mutable
{
  using ArgsT = std::tuple<decltype(args)...>;
  if constexpr (std::is_same_v<ArgsT, std::tuple<int, int>>) {
   // two integers = add to total
   auto [a, b] = ArgsT(args...);
   total += a + b;
  } else if constexpr (std::is_same_v<ArgsT, std::tuple<>>) {
   // no parameters = print
   print(total);
  } else {
   static_assert(!sizeof(Op*), "Don't know what you are asking me to do.");
  }
};
```

This might come in handy if you have a lambda that is used to accumulate something: You can pass the lambda to the function that expects to do the accumulating, and then call the lambda using a secret knock to extract the answer.

```
auto lambda = [limit, total = 0](auto value) mutable
{
 using T = decltype(value);
 if constexpr (std::is_same_v<T, const char*>) {
  // secret knock: Return total if invoked with const char*
  return total;
 } else {
  // Otherwise, just add them up until we hit the limit.
  total += value;
  return total <= limit;
 }
};

auto unused = std::find_if_not(begin, end, std::ref(lambda));
if (unused != end) print("Limit exceeded.");
auto total = lambda("total"); // extract the total
```

This is basically a complete and utter abuse of the language, and I hope you're ashamed of yourself.

Bonus chatter: All of this is just a way of defining a `struct` without having to say the word `struct`.

```
struct
{
 double limit;
 double total = 0.00;

 auto add_tax(auto tax_rate) { total += total * tax_rate; }
 auto apply_discount(auto amount) { total -= std::max(amount, total); }
 auto get_total() const { return total; }
} lambda{1000.00 /* limit */};
```

Bonus bonus chatter: Java anonymous classes provide a more straightforward syntax:

```
var lambda = new Object() {
  int total = 0;
  public void add(int value) { total += value; }
  public int get_total() { return total; }
};

lambda.add(2);
lambda.add(3);
var result = lambda.get_total();
```

Raymond Chen

Follow