【C++】Day83

Class	C++
Date	@April 24, 2022
Material	
Series Number	
Summary	Variadic Templates

【Ch16】Templates and Generic Programming

16.3 Variadic Templates

A variadic template is a template function or class that can take a varying number of parameters. The varying parameters are known as a parameter pack.

There are two kinds of parameter packs: A template parameter pack represents zero or more template parameters, and a function parameter pack represents zero or more function parameters.

For example:

```
// Args is a template parameter pack; rest is a function parameter pack
// Args represents zero or more template type parameters
// rest represents zero or more function parameters
template <typename T, typename ... Args>
void foo(const T &t, const Args& ... rest);
```

declares that `foo` is a variadic function that has one type parameter named `T` and a template parameter pack named `Args`.

For a variadic template, the compiler deduces the number and types of parameters in the pack.

For example, given these calls:

```
int i = 0; double d = 3.24; string s = "Hi";
foo(i, s, 42, d); // Three parameters in the pack
foo(s, 42, "hi"); // Two parameters in the pack
foo(d, s); // One parameter in the pack
foo("hi"); // Zero parameter in the pack
```

The sizeof Operator

When we need to know how many elements there are in a pack, we can use the `sizeof...` operator.

```
template <typename ... Args> void g (Args ... args) {
  std::cout << sizeof...(Args) << std::endl;
  std::cout << sizeof...(args) << std::endl;
}
```

16.4.1 Writing a Variadic Function Template

We could use an `initializer_list` to define a function that can take a varying number of arguments. However, the arguments must have the same type.

Variadic functions are used when we know neither the number nor the types of the arguments we want to process.

As an example, we will define a function like our earlier `error_msg` function.

Variadic functions are often recursive. The first call processes the first argument in the pack and calls itself on the remaining arguments.

To stop the recursion, we also need to define a nonvariadic `print` function that will take a stream and an object.

```
// Function to end the recursion and print the last element
// This function must be declared before the variadic version of print is defined
template <typename T> std::ostream &print(std::ostream &os, const T& t) {
  return os << t;
}

// This version of print will be called for all but the last element in the pack
template <typename T, typename ... Args> std::sotream &print(std::ostream &os, const T& t, const Args& ... rest) {
  os << t << ", ";
  return print(os, rest...);
}
```

The first version of `print` stops the recursion and prints the last argument in the initial call to print.

The key part is the call to `print` inside the variadic function:

```
return print(os, rest...);
```

The first argument in `rest` will be bound to `t`. The remaining arguments in `rest` from the parameter pack for the next call to `print`.

Warning: A declaration for the nonvariadic version of print must be in scope when the variadic version is defined.

16.4.2 Pack Expansion

Aside from taking its size, the only other thing we can do with a parameter pack is to expand it.

When we expand a pack, we also provide a pattern to be used on each expanded element.

Expanding a pack separates the pack into its constituent elements, applying the pattern to each element as it does so. We trigger an expansion by putting an ellipsis(`...`) to the right of the pattern.

For example, our `print` function contains two expansions:

```
template <typename T, typename ... Args>
std::ostream &print(std::ostream &os, const T &t, const Args& ... rest) { // Expand Args
  os << t << ", ";
  return print(os, rest...); // Expand rest
}
```

The first expansion expands the template parameter pack and generates the function parameter list for `print`.

The second expansion appears in the call to `print`. That pattern generates the argument list for the call to `print`.

The expansion of `Args` applies the pattern `const Args&` to each element in the template parameter pack `Args`. The expansion of this pattern is a comma-separated list of zero or more parameter types, each of which will have the form `const type&`.

Understanding Pack Expansions

We can also write a second variadic function that calls debug_rep on each of its arguments and then calls print to print the resulting strings:

```
// Call debug_rep on each argument in the call to print
template <typename ... Args>
std:;ostream &errorMsg(std::ostream &os, const Args& ... rest) {
  return print(os, debug_rep(rest)...);
}
```

This call to `print` uses the pattern `debug_rep(rest)`.