templates

part 4

variadic function templates (from C++ Primer)

variadic templates and parameter packs

a variadic template is a template that can take a varying number of parameters, which are known as a template **parameter pack**

a parameter pack represents **zero or more** parameters, which can be template parameters or function parameters

an ellipsis (...) to the left of a name is used to indicate that it represents a pack

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

pack expansion

an ellipsis to the right of a name is used to trigger **pack expansion**, where a **pattern** is applied to each element in the pack

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

in the `foo` function declaration, the parameter pack `Args` is expanded and the pattern `const Args&` is applied to each element to generate the function parameter list

more on parameter packs: https://en.cppreference.com/w/cpp/language/parameter_pack

variadic function template example

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

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

```
int i = 0; double d = 3.14; std::string s = “how now brown cow”;
foo(i, s, 42, d);        // three parameters in pack
foo(s, 42, “hi”);        // two parameters in pack
food(d, s);              // one parameter in pack
foo(“hi”);               // empty pack
```

the compiler will deduce types and instantiate four different instances of `foo`:

```
void foo(const int&, const std::string&, const int&, const double&);
void foo(const std::string&, const int&, const char(&)[3]);
void foo(const double&, const std::string&);
void foo(const char(&)[3]);
```

the `sizeof...` operator

we can use the `sizeof...` operator to see how many elements are in a pack:

```
template <typename ... Args>
void g(Args ... f_args)
{
    std::cout << sizeof...(Args) << std::endl;       // number of template parameters
    std::cout << sizeof...(f_args) << std::endl;     // number of function parameters
}
```

note that there is no simple way to iterate over a parameter pack!

writing a variadic function template

variadic functions are useful when we know **neither the number nor the types** of the arguments we want to process, and are often recursive

```
// 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>
ostream &print(ostream &os, const T &t)
{
    return os << t; // no separator after the last element in the pack
}

// this version of print will be called for all but the last element in the pack
template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
    os << t << ", "; // print the first argument with separator
    return print(os, rest...); // recursive call; print the other arguments
}
```

recursive variadic function execution

```
// nonvariadic version
template <typename T> ostream &print(ostream &os, const T &t)
{ return os << t; // no separator after the last element in the pack }

// variadic version
template <typename T, typename... Args> ostream &print(ostream &os, const T &t, const Args&... rest)
{ os << t << ", "; // print the first argument with separator
  return print(os, rest...); // recursive call; print the other arguments }

print(cout, i, s, 42) // two parameters in the pack
```

for the last call, both versions of the function are viable and equally good matches, but the nonvariadic template is more specialized so it is used

call	`t`	`rest...`
`print(cout, i, s, 42)`	`i`	`s, 42`
`print(cout, s, 42)`	`s`	`42`
`print(cout, 42)`	calls the nonvariadic version of `print`	

variadic data structures

from: https://riptutorial.com/cplusplus/example/19276/variadic-template-data-structures

variadic data structure example

```
// general (empty) definition
template<typename ... T>
struct Shuple {};

// recursive case specialization
template<typename T, typename ... Rest>
struct Shuple<T, Rest ...> {
    Shuple(const T& first, const Rest& ... rest): first(first), rest(rest...) {}

    T first;
    Shuple<Rest ... > rest;
};

Shuple<int, float> data;
```

the declaration for `data` creates the following structs (ignoring constructors):

```
struct Shuple<int, float> { int first; Shuple<float> rest; };
struct Shuple<float> { float first; Shuple<> rest; };
struct Shuple<> {};
```

variadic data structure example

for this to be remotely useful, we must add a method to access elements:

```
template<typename T, typename ... Rest>
struct Shuple<T, Rest ...>
{
    ...
    template<size_t idx>      // get is templated on the index of the member to access
    auto get()                // usage will look like data.get<1>()
    {
        return GetHelper<idx, Shuple<T,Rest...>>::get(*this); // actual work done by helper class
    }
    ...
};
```

we can’t define the functionality directly in `Shuple`’s `get` because we would need to specialize on `idx`, but it is not possible to specialize a template member function without specializing the containing class template

variadic data structure example

```
template<size_t idx, typename T> struct GetHelper;

// base case (idx == 0)
template<typename T, typename ... Rest>
struct GetHelper<0, Shuple<T, Rest ... >>
{
    static T get(Shuple<T, Rest...>& data)
    {
        return data.first;
    }
};

// recursive case
template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, Shuple<T, Rest ... >>
{
    static auto get(Shuple<T, Rest...>& data)
    {
        return GetHelper<idx-1, Shuple<Rest ...>>::get(data.rest); // decrement idx
    }
};
```

variadic data structure example

tracing the behavior of `get`:

```
Shuple<int, float> data(1, 2.1);
data.get<1>();
\\ invokes
GetHelper<1, Shuple<int, float>>::get(data)
\\ which in turn invokes
GetHelper<0, Shuple<float>>::get(data.rest)
\\ which returns
data.rest.first // 2.1
```

more interesting reading on variadic templates:

https://eli.thegreenplace.net/2014/variadic-templates-in-c/