A Brief Introduction to Variadic Templates

Author: Douglas Gregor, Indiana University
Document number: N2087=06-0157
Date: September 10, 2006
Project: Programming Language C++, Evolution Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

When a C++ template is defined, it is given a fixed number of template parameters that must be specified when the template is used. *Variadic templates* provide templates with the ability to accept an arbitrary number of template arguments, facilitating clean implementations of type-safe printf(), "inherited" constructors, and class templates such as tuple (a generalized pair that stores any number of values).[1] This introduction illustrates the use of variadic templates to implement a simple, type-safe printf(), with no unsightly macros. For instance, the following code should compile and execute correctly with our printf(). Note that making this call using the current C/C++ printf() invokes undefined behavior, because it can't handle std::strings.

```
const char* msg = "The value of %s is about %g (unless you live in %s).\n";
printf(msg, std::string("pi"), 3.14159, "Indiana");[2]
```

C's printf() is implemented using the ellipsis (...) to allow a variable number of function arguments. Variadic templates also use the ellipsis syntax, but to describe a new kind of entity: a *parameter pack*. The following tuple template declares a *template type parameter pack* named Args:

```
template<typename... Args> class tuple { /* implementation */ };
```

Tuple can be instantiated with any number of types, e.g., tuple<int, float, std::string>, tuple<int>, or even tuple<float>. These arguments will be packed into Args.

To build our type-safe printf(), we use the following strategy: write out the string up until the first value is reached, print that value, then call printf() recursively to print the rest of the string and remaining values. The entire template-recursive function follows:

```
template<typename T, typename... Args>
void printf(const char* s, const T& value, const Args&... args) {
  while (*s) {
    if (*s == '%' && *++s != '%') {
      // ignore the character that follows the '%': we already know the type!
      std::cout << value;
      return printf(++s, args...);
    }
    std::cout << *s++;
  }
  throw std::runtime_error("extra arguments provided to printf");
}
```

There is a bunch of new syntax in this example, so we'll break it into pieces. As with the tuple template, printf() contains a template type parameter pack named Args. The template parameter T, on the other hand, is just a normal template type parameter. That means that we can instantiate printf() with one or more types, e.g., printf<std::string, double>() (so T = std::string and Args would contain double) or printf<int, float, double>() (so T = int and Args would contain float and double).

Next, we move on to the declaration of our type-safe printf():

```
template<typename T, typename... Args>
void printf(const char* s, const T& value, const Args&... args);
```

[1] Implementations currently emulate variadic templates, but the emulation is quite cumbersome and very limited.
[2] See `http://www.agecon.purdue.edu/crd/Localgov/Second%20Level%20pages/Indiana_Pi_Story.htm` for an explanation.

The function parameter list of printf uses the ellipsis again, to create another kind of parameter pack. The ellipsis here states that args is a *function parameter pack*. Function parameter packs are to function parameters as template parameter packs are to template parameters.

Note that the declaration of the function parameter pack args actually uses the template parameter pack Args as part of its type. This sets up a relationship between the two parameter packs, because the i^{th} parameter in args, args_i, will have the type Args_i const& (where Args_i is the i^{th} type in the template parameter pack Args). For instance, the type of the expression &printf<int, float, double> will be:

```
void (*)(const char*, const int&, const float&, const double&)
```

Template argument deduction works with function parameter packs in the obvious way. In the call printf(msg, std::string("pi"), 3.14159, "Indiana"), we deduce T to the argument std::string and Args to the arguments double and const char*. What's most interesting is how printf() deals with those arguments. It loops through the string printing characters until it hits a percent sign. Then it prints out the current value and recurses with the following return statement:

```
return printf(++s, args...);
```

The ellipsis in this call is a *meta operator* that unpacks the function parameter pack args into separate arguments to printf(). The argument args... expands to all of the values that were packed into args when printf() was called. This call will go to either the printf() template (if args has at least one parameter) or to the non-template printf() below, which completes the recursion. The ellipsis therefore has two roles: when it occurs *to the left* of the name of a parameter, it declares a parameter pack; when it occurs *to the right* of a template or function call argument, it unpacks the parameter packs into separate arguments.[3]

```
void printf(const char* s) {
  while (*s) {
    if (*s == '%' && *++s != '%')
      throw std::runtime_error("invalid format string: missing arguments");
    std::cout << *s++;
  }
}
```

Let's step back and walk through our original call: printf(msg, std::string("pi"), 3.14159, "Indiana");

At this call, template argument deduction determines that T=std::string and Args contains double and const char*. The call proceeds with value=std::string("pi") and args containing 3.14159 and "Indiana". When we instantiate this printf(), it prints "The value of " followed by "pi", then the recursive call to printf() passes on just the arguments packed into args, which is (effectively):

```
printf(s, 3.14159, "Indiana");
```

Now, template argument deduction determines that T=double and Args contains const char*. The call proceeds with value=3.14159 and args containing "Indiana", printing "is about " followed by 3.14159. Now our recursive call is effectively:

```
printf(s, "Indiana");
```

This time, T=const char* and Args is empty. The call proceeds with value="Indiana" and args is also empty, and we print "(unless you live in ", followed by "Indiana". Our final recursive call boils down to:

```
printf(s);
```

At this point, our recursion will terminate, because args is completely empty. The non-template printf() will print the remainder of the string.

With variadic templates, we were able to define a type-safe printf() in just a few simple lines of code; with slightly more effort, we could handle justification, precision and padding, warn about problems with incorrect format strings (e.g., "%f" vs. "%i"), or create something better that omits the need for formatting strings entirely. Variadic templates are about much more than just a type-safe printf(). They offer a concise, self-contained mechanism that simplifies many modern C++ libraries (including TR1), greatly reducing the need for extra-linguistic preprocessor manipulation.

[3] The use of the ellipsis for unpacking is needed to eliminate ambiguities; see the full paper for an explanation.