Variadic template

From Wikipedia, the free encyclopedia

In computer programming, variadic templates are templates that take a variable number of arguments.

Variadic templates are supported by C++ (since the C++11 standard), and the D programming language.

Contents

C++

The variadic template feature of C++ was designed by Douglas Gregor and Jaakko Järvi [1][2] and was later standardized in C++11. Prior to C++11, templates (classes and functions) could only take a fixed number of arguments, which had to be specified when a template was first declared. C++11 allows template definitions to take an arbitrary number of arguments of any type.

```
template<typename... Values> class tuple;
```

The above template class `tuple` will take any number of typenames as its template parameters. Here, an instance of the above template class is instantiated with four type arguments:

```
tuple<int, std::vector<int>, std::map<std::string, std::vector<int>>> some_instance_name;
```

The number of arguments can be zero, so `tuple<> some_instance_name;` will work as well.

If one does not want to have a variadic template that takes 0 arguments, then this definition will work as well:

```
template<typename First, typename... Rest> class tuple;
```

Variadic templates may also apply to functions, thus not only providing a type-safe add-on to variadic functions (such as printf) - but also allowing a printf-like function to process non-trivial objects.

```
template<typename... Params> void printf(const std::string &str_format, Params... parameters);
```

The ellipsis (...) operator has two roles. When it occurs to the left of the name of a parameter, it declares a parameter pack. Using the parameter pack, the user can bind zero or more arguments to the variadic template parameters. Parameter packs can also be used for non-type parameters. By contrast, when the ellipsis operator occurs to the right of a template or function call argument, it unpacks the parameter packs into separate arguments, like the `args...` in the body of `printf` below. In practice, the use of an ellipsis operator in the code causes the whole expression that precedes the ellipsis to be repeated for every subsequent argument unpacked from the argument pack; and all these expressions will be separated by a comma.

The use of variadic templates is often recursive. The variadic parameters themselves are not readily available to the implementation of a function or class. Therefore, the typical mechanism for defining something like a C++11 variadic `printf` replacement would be as follows:

```
void printf(const char *s)
{
  while (*s) {
    if (*s == '%') {
      if (*(s + 1) == '%') {
        ++s;
      }
      else {
        throw std::runtime_error("invalid format string: missing arguments");
      }
    }
    std::cout << *s++;
  }
}
```

```
template<typename T, typename... Args>
void printf(const char *s, T value, Args... args)
{
  while (*s) {
    if (*s == '%') {
      if (*(s + 1) == '%') {
        ++s;
      }
      else {
        std::cout << value;
        s += 2; // this only works on 2 characters format strings ( %d, %f, etc ). Fails miserably with %5.4f
        printf(s, args...); // call even when *s == 0 to detect extra arguments
        return;
      }
    }
    std::cout << *s++;
  }
}
```

This is a recursive template. Notice that the variadic template version of `printf` calls itself, or (in the event that `args...` is empty) calls the base case.

There is no simple mechanism to iterate over the values of the variadic template. There are few ways to translate the argument pack into single argument use. Usually this will rely on function overloading, or - if the function can simply pick one argument at a time - using a dumb expansion marker:

```
template<typename... Args> inline void pass(Args&&...) {}
```

which can be used as follows:

```
template<typename... Args> inline void expand(Args&&... args) {
  pass( some_function(args)... );
}

expand(42, "answer", true);
```

which will expand to something like:

```
pass( some_function(arg1), some_function(arg2), some_function(arg3) etc... );
```

The use of this "pass" function is necessary, since the expansion of the argument pack proceeds by separating the function call arguments by commas, which are not equivalent to the comma operator. Therefore, `some_function(args)...;` will never work. Moreover, this above solution will only work when the return type of `some_function` is not `void`. Furthermore, the `some_function` calls will be executed in an unspecified order, because the order of evaluation of function arguments is undefined. To avoid the unspecified order, brace-enclosed initializer lists can be used, which guarantee strict left-to-right order of evaluation. To avoid the need for a not `void` return type, the comma operator can be used to always yield `1` in each expansion element.

```
struct pass {
  template<typename ...T> pass(T...) {}
};

pass{(some_function(args), 1)...};
```

Instead of executing a function, a lambda expression may be specified and executed in place, which allows executing arbitrary sequences of statements in-place.

```
pass{([&](){ std::cout << args << std::endl; }(), 1)...};
```

However, in this particular example, a lambda function is not necessary. A more ordinary expression can be used instead:

```
pass{(std::cout << args << std::endl, 1)...};
```

Another way is to use overloading with "termination versions" of functions. This is more universal, but requires a bit more code and more effort to create. One function receives one argument of some type *and* the argument pack, whereas the other receives neither. (If both have the same list of initial parameters, the call would be ambiguous - a variadic parameter pack alone cannot disambiguate a call.) For example:

```
void func() {} // termination version

template<typename Arg1, typename... Args>
void func(const Arg1& arg1, const Args&... args)
{
  process( arg1 );
  func(args...); // note: arg1 does not appear here!
```

```
}
```

If args... contains at least one argument, it will redirect to the second version - a parameter pack can be empty, in which case it will simply redirect to the termination version, which will do nothing.

Variadic templates can also be used in an exception specification, a base class list, or the initialization list of a constructor. For example, a class can specify the following:

```
template <typename... BaseClasses> class ClassName : public BaseClasses... {
public:

  ClassName (BaseClasses&&... base_classes) : BaseClasses(base_classes)... {}
};
```

The unpack operator will replicate the types for the base classes of ClassName, such that this class will be derived from each of the types passed in. Also, the constructor must take a reference to each base class, so as to initialize the base classes of ClassName.

With regard to function templates, the variadic parameters can be forwarded. When combined with universal references (see above), this allows for perfect forwarding:

```
template<typename TypeToConstruct> struct SharedPtrAllocator {

  template<typename ...Args> std::shared_ptr<TypeToConstruct> construct_with_shared_ptr(Args&&... params) {
    return std::shared_ptr<TypeToConstruct>(new TypeToConstruct(std::forward<Args>(params)...));
  }
};
```

This unpacks the argument list into the constructor of TypeToConstruct. The std::forward<Args>(params) syntax is the syntax that perfectly forwards arguments as their proper types, even with regard to rvalue-ness, to the constructor. The unpack operator will propagate the forwarding syntax to each parameter. This particular factory function automatically wraps the allocated memory in a std::shared_ptr for a degree of safety with regard to memory leaks.

Additionally, the number of arguments in a template parameter pack can be determined as follows:

```
template<typename ...Args> struct SomeStruct {
  static const int size = sizeof...(Args);
};
```

The expression SomeStruct<Type1, Type2>::size will yield 2, while SomeStruct<>::size will give 0.

D

Definition

Definition of variadic templates in D are based on their C++ counterpart:

```
template VariadicTemplate(Args...) { /* Body */ }
```

Likewise, any argument can precede the argument list:

```
template VariadicTemplate(T, string value, alias symbol, Args...) { /* Body */ }
```

Basic usage

Variadic arguments are very similar to constant array in their usage. They can be iterated upon, accessed by an index, have a length property, and can be sliced. Operations are interpreted at compile time, which means operands can't be runtime value (such as function parameters).

Anything which is known at compile time can be passed as a variadic arguments. It makes variadic arguments similar to template alias arguments (http://dlang.org/template#TemplateAliasParameter), but more powerful, as they also accept basic types (char, short, int...).

Here is an example that print the string representation of the variadic parameters. StringOf and StringOf2 produce equal results.

```
static int s_int;

struct Dummy {}

void main() {
  pragma(msg, StringOf!("Hello world", uint, Dummy, 42, s_int));
  pragma(msg, StringOf2!("Hello world", uint, Dummy, 42, s_int));
}
```

```
template StringOf(Args...) {
  enum StringOf = Args[0].stringof ~ StringOf!(Args[1..$]);
}

template StringOf() {
  enum StringOf = "";
}

template StringOf2(Args...) {
  static if (Args.length == 0)
    enum StringOf2 = "";
  else
    enum StringOf2 = Args[0].stringof ~ StringOf2!(Args[1..$]);
}
```

Outputs:

```
"Hello world"uintDummy42s_int
"Hello world"uintDummy42s_int
```

TypeTuple

Variadic template are often used to create a construction called TypeTuple (see std.typetuple (http://dlang.org/phobos/std_typetuple.html#.TypeTuple)). A TypeTuple definition is actually very straightforward:

```
template TypeTuple(Args...) {
  alias TypeTuple = Args;
}
```

This structure allows one to manipulate a "list" of variadic arguments that will auto expand. This enables any operation you would expect:

```
import std.typetuple;

void main() {
  // Note: TypeTuple can't be modified, and an alias can't be rebound, so we'll need to define new names for our modifications.
  alias numbers = TypeTuple!(1, 2, 3, 4, 5, 6);
  // Slicing
  alias lastHalf = numbers[$ / 2 .. $];
  static assert(lastHalf == TypeTuple!(4, 5, 6));
  // TypeTuple auto expansion (thanks to D eponymous template feature).
  alias digits = TypeTuple!(0, numbers, 7, 8, 9);
  static assert(digits == TypeTuple!(0, 1, 2, 3, 4, 5, 6, 7, 8, 9));
  // std.typetuple provides templates to work with TypeTuple, such as anySatisfy, allSatisfy, staticMap, and Filter.
  alias evenNumbers = Filter!(isEven, digits);
  static assert(evenNumbers == TypeTuple!(0, 2, 4, 6, 8));
}

template isEven(int number) {
  enum isEven = (0 == (number % 2));
}
```

See also

For articles on variadic constructs other than templates

- Variadic function
- Variadic macro in the C preprocessor

References

1. Douglas Gregor & Jaakko Järvi. " "Variadic Templates for C++0x", in Journal of Object Technology, vol. 7, no. 2, Special Issue OOPS Track at SAC 2007, February 2008, pp. 31-51".
2. Douglas Gregor; Jaakko Järvi & Gary Powell. (February 2004). " "Variadic templates. Number N1603=04-0043 in ISO C++ Standard Committee Pre-Sydney mailing" ".

External links

- Working draft for the C++ language, January 16, 2012 (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf)
- Variadic Templates in D language (http://dlang.org/variadic-function-templates.html)