Variadic Templates

Venkatesh

WDC

October 24, 2017

Outline

1. Introduction
2. Fundamentals
   - Parameter Packs
3. Examples
   - Expansion loci
   - Variadic Function template
   - Tuples

Introduction

Variadic templates

mechanism for passing an arbitrary number of arguments of arbitrary types to a template

Variadic class template

e.g:

```
template<class ... Types> struct Tuple {};
Tuple<> t0;           // Types contains no arguments
Tuple<int> t1;        // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and
    float
Tuple<0> error;       // error: 0 is not a type
```

Variadic function template

e.g:

```
template<class ... Types> void f(Types ... args);
f();       // OK: args contains no arguments
f(1);      // OK: args contains one argument: int
f(2, 1.0); // OK: args contains two arguments: int and double
```

```
template <typename... Ts>
class C
{
    // fill the body
};

template <typename... Ts>
void fun(const Ts&... vs)
{
    // fill the body
}
```

A New Kind: Parameter Packs

Ts is not a type; vs is not a value!

typedef Ts MyList; *// error!*
Ts var; *// error!*
auto copy = vs; *// error!*

- Ts is an alias for a list of types
- vs is an alias for a list of values
- Either list may be potentially empty
- Both obey only specific actions

Parameter pack

template parameter pack

is a template parameter that accepts zero or more template arguments

function parameter pack

is a function parameter that accepts zero or more function arguments

Parameter pack Syntax

Template parameter pack

type ... Args(optional)

typename|**class** ... Args(optional)

Function parameter pack

Args ... args(optional)

Constraint

template parameter pack must be the final parameter in the template parameter list

e.g:

```
template<typename... Ts, typename U> struct Invalid;
// Error: Ts.. not at the end
```

Using parameter packs

- Apply sizeof... to it, this will return how many types in Ts

  size_t items = **sizeof**...(Ts); *// or vs*

- Parameter pack expansion

Syntax

pattern ...

expands to comma-separated list of zero or more patterns. Pattern must include at least one parameter pack

Pack expansion

e.g:

```
template<class ...Us> void f(Us... pargs) {}
template<class ...Ts> void g(Ts... args) {
    f(&args...); // "&args..." is a pack expansion
                 // "&args" is its pattern
}
g(1, 0.2, "a");
// Ts... args expand to int E1, double E2, const char* E3
// &args... expands to &E1, &E2, &E3
// Us... pargs expand to int* E1, double* E2, const char** E3
```

Expansion rules

Use	Expansion
Ts...	T1, .., Tn
Ts&&...	T1&&, .., Tn&&
x<Ts,Y>::z...	x<T1,Y>::z, .., x<Tn,Y>::z
x<Ts&,Us>...	x<T1&,U1>, .., x<Tn&,Un>
func(5,vs)...	func(5,v1), .., func(5,vn)

- When expanding two lists in lock-step, they should have the same size.

Multiple Expansions

Expansion proceeds inwards outwards

```
template <class ... Ts> void fun(Ts... vs) {

    gun(A<Ts...>::hun(vs)...);
    gun(A<Ts...>::hun(vs...));
    gun(A<Ts>::hun(vs)...);

}
```

These are different expansions

Expansion loci

1. Initializer lists

any a[] = { vs... };

example

```
template<typename... Ts> void func(Ts... args){
    const int size = sizeof...(args) + 2;
    int res[size] = {1,args...,2};
    int dummy[sizeof...(Ts)] = { (std::cout << args, 0)... };
}
```

Expansion loci

2. Function argument lists (simply function-call operator)

example

f(&args...); *// expands to f(&E1, &E2, &E3)*
f(n, ++args...); *// expands to f(n, ++E1, ++E2, ++E3);*
f(++args..., n); *// expands to f(++E1, ++E2, ++E3, n);*
f(h(args...) + args...); *// expands to*
// f(h(E1,E2,E3) + E1, h(E1,E2,E3) + E2, h(E1,E2,E3) + E3)

Expansion loci

3. Template argument lists

example

```
template<class A, class B, class...C> void func(A arg1, B
    arg2, C...arg3)
{
    container<A,B,C...> t1;
    // expands to container<A,B,E1,E2,E3>
    container<C...,A,B> t2;
    // expands to container<E1,E2,E3,A,B>
    container<A,C...,B> t3;
    // expands to container<A,E1,E2,E3,B>
}
```

Expansion loci

4. Base specifiers and member initializer lists

example

```
template <typename... Ts>
struct C : Ts... {};

template <typename... Ts>
struct D : Box<Ts>... {  };

template<class... Mixins>
class X : public Mixins... {
 public:
   X(const Mixins&... mixins) : Mixins(mixins)... { }
};
```

Expansion loci

5. Lambda captures

example

```
template<class ...Args>
void f(Args... args) {
    auto lm = [&, args...] { return g(args...); };
    lm();
}
```

Expansion loci

6. Function parameter list

example

```
template<typename ...Ts> void f(Ts...) {}
f('a', 1);  // Ts... expands to void f(char, int)
f(0.1);     // Ts... expands to void f(double)
```

How to use variadic templates

usage approach

Pattern Matching!

Examples : User defined Tuple

```
Tuple<double , int, char> x {1.1, 42, 'a'};
cout << x << "\n";
cout << get<1>(x) << "\n";
```

Advantages

- Library utilities std::make_unique, std::make_shared

std::make_unique for single objects

```
template<typename Tp, typename... Args>
make_unique(Args&&... args) {
    return unique_ptr<Tp>(new
  Tp(std::forward<Args>(args)...));
}
```

References

The C++ Programming Language [4th Edition] - Bjarne Stroustrup

Thank You