Parameter pack

A template parameter pack is a template parameter that accepts zero or more template arguments (non-types, types, or templates). A function parameter pack is a function parameter that accepts zero or more function arguments.

A template with at least one parameter pack is called a *variadic template*.

Syntax

Template parameter pack (appears in a class template and in a function template parameter list)

type **...** *Args*(optional)	(1)	(since C++11)
typename|**class ...** *Args*(optional)	(2)	(since C++11)
template < *parameter-list* **> typename**(C++17)**|class ...** *Args*(optional)	(3)	(since C++11)

Function parameter pack (a form of declarator, appears in a function parameter list of a variadic function template)

Args **...** *args*(optional)	(4)	(since C++11)

Parameter pack expansion (appears in a body of a variadic template)

pattern **...**	(5)	(since C++11)

1) A non-type template parameter pack with an optional name
2) A type template parameter pack with an optional name
3) A template template parameter pack with an optional name
4) A function parameter pack with an optional name
5) Parameter pack expansion: expands to comma-separated list of zero or more `patterns`. Pattern must include at least one parameter pack.

Explanation

A variadic class template can be instantiated with any number of template arguments:

```
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
```

A variadic function template can be called with any number of function arguments (the template arguments are deduced through template argument deduction):

```
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
```

In a primary class template, the template parameter pack must be the final parameter in the template parameter list. In a function template, the template parameter pack may appear earlier in the list provided that all following parameters can be deduced from the function arguments, or have default arguments:

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

template<typename ...Ts, typename U, typename=void>
void valid(U, Ts...);     // OK: can deduce U
// void valid(Ts..., U);  // Can't be used: Ts... is a non-deduced context in this position

valid(1.0, 1, 2, 3);      // OK: deduces U as double, Ts as {int,int,int}
```

Pack expansion

A pattern followed by an ellipsis, in which the name of at least one parameter pack appears at least once, is *expanded* into zero or more comma-separated instantiations of the pattern, where the name of the parameter pack is replaced by each of the elements from the pack, in order.

```
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
```

If the names of two parameter packs appear in the same pattern, they are expanded simultaneously, and they must have the same length:

```
template<typename...> struct Tuple {};
template<typename T1, typename T2> struct Pair {};

template<class ...Args1> struct zip {
    template<class ...Args2> struct with {
        typedef Tuple<Pair<Args1, Args2>...> type;
//        Pair<Args1, Args2>... is the pack expansion
//        Pair<Args1, Args2> is the pattern
    };
};

typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
// Pair<Args1, Args2>... expands to
```

```
// Pair<short, unsigned short>, Pair<int, unsigned int>
// T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>

typedef zip<short>::with<unsigned short, unsigned>::type T2;
// error: pack expansion contains parameter packs of different lengths
```

If a pack expansion is nested within another pack expansion, the parameter packs that appear inside the innermost pack expansion are expanded by it, and there must be another pack mentioned in the enclosing pack expansion, but not in the innermost one:

```
template<class ...Args>
    void g(Args... args) {
        f(const_cast<const Args*>(&args)...);
 // const_cast<const Args*>(&args) is the pattern, it expands two packs
 // (Args and args) simultaneously

        f(h(args...) + args...); // Nested pack expansion:
   // inner pack expansion is "args...", it is expanded first
   // outer pack expansion is h(E1, E2, E3) + args..., it is expanded
   // second (as h(E1,E2,E3) + E1, h(E1,E2,E3) + E2, h(E1,E2,E3) + E3)
}
```

Expansion loci

Depending on where the expansion takes place, the resulting comma-separated list is a different kind of list: function parameter list, member initializer list, attribute list, etc. The following is the list of all allowed contexts

Function argument lists

A pack expansion may appear inside the parentheses of a function call operator, in which case the largest expression to the left of the ellipsis is the pattern that is expanded.

```
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(const_cast<const Args*>(&args)...);
// f(const_cast<const E1*>(&X1), const_cast<const E2*>(&X2), const_cast<const E3*>(&X3))
f(h(args...) + args...); // expands to
// f(h(E1,E2,E3) + E1, h(E1,E2,E3) + E2, h(E1,E2,E3) + E3)
```

Template argument lists

Pack expansions can be used anywhere in a template argument list, provided the template has the parameters to match the expansion.

```
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>
}
```

Function parameter list

In a function parameter list, if an ellipsis appears in a parameter declaration (whether it names a function parameter pack (as in, *Args* **...** *args*) or not) the parameter declaration is the pattern:

```
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)

template<typename ...Ts, int... N> void g(Ts (&...arr)[N]) {}
int n[1];
g<const char, int>("a", n); // Ts (&...arr)[N] expands to
                            // const char (&)[2], int(&)[1]
```

Note: In the pattern `Ts (&...arr)[N]`, the ellipsis is the innermost element, not the last element as in all other pack expansions.

Note: `Ts (&...)[N]` is not allowed because the C++11 grammar requires the parenthesized ellipsis to have a name: CWG #1488 (http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_active.html#1488) .

Template parameter list

Pack expansion may appear in a template parameter list:

```
template<typename... T> struct value_holder
{
    template<T... Values> // expands to a non-type template parameter
    struct apply { };     // list, such as <int, char, int(&)[5]>
};
```

Base specifiers and member initializer lists

A pack expansion may designate the list of base classes in a class declaration. Typically, this also means that the constructor needs to use a pack expansion in the member initializer list to call the constructors of these bases:

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

Braced init lists

In a braced-init-list (brace-enclosed list of initializers and other braced-init-lists, used in list-initialization and some other contexts), a pack expansion may appear as well:

```
template<typename... Ts> void func(Ts... args){
    const int size = sizeof...(args) + 2;
    int res[size] = {1,args...,2};
    // since initializer lists guarantee sequencing, this can be used to
    // call a function on each element of a pack, in order:
    int dummy[sizeof...(Ts)] = { (std::cout << args, 0)... };
}
```

Lambda captures

A parameter pack may appear in the capture clause of a lambda expression

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

The sizeof... operator

The sizeof... operator is classified as a pack expansion as well

```
template<class... Types>
struct count {
    static const std::size_t value = sizeof...(Types);
};
```

Dynamic exception specifications

The list of exceptions in a dynamic exception specification may also be a pack expansion

```
template<class...X> void func(int arg) throw(X...)
{
 // ... throw different Xs in different situations
}
```

Alignment specifier

Pack expansions are allowed in both the lists of types and the lists of expressions used by the keyword alignas

Attribute list

Pack expansions are allowed in the lists of attributes, as in `[[attributes...]]`. For example: `void [[attributes...]] function()`

Fold-expressions

In fold-expressions, the pattern is the entire subexpression that does not contain an unexpanded parameter pack.

Using-declarations

In using declaration, ellipsis may appear in the list of declarators, this is useful when deriving from a parameter pack: (since C++17)

```
template <typename... bases>
struct X : bases... {
        using bases::g...;
};
X<B, D> x; // OK: B::g and D::g introduced
```

Notes

This section is incomplete
Reason: a few words about partial specializations and other ways to access individual elements?
Mention recursion vs logarithmic vs shortcuts such as fold expressions

Example

Run this code

```
#include <iostream>

void tprintf(const char* format) // base function
{
    std::cout << format;
}

template<typename T, typename... Targs>
void tprintf(const char* format, T value, Targs... Fargs) // recursive variadic function
{
    for ( ; *format != '\0'; format++ ) {
        if ( *format == '%' ) {
           std::cout << value;
           tprintf(format+1, Fargs...); // recursive call
           return;
        }
        std::cout << *format;
    }
}

int main()
{
    tprintf("% world% %\n","Hello",'!',123);
```

```
    return 0;
}
```

Output:

```
Hello world! 123
```

The above example defines a function similar to `std::printf`, that replace each occurrence of the character % in the format string with a value.

The first overload is called when only the format string is passed and there is no parameter expansion.

The second overload contains a separate template parameter for the head of the arguments and a parameter pack, this allows the recursive call to pass only the tail of the parameters until it becomes empty.

Targs is the template parameter pack and **Fargs** is the function parameter pack

See also

function template	
class template	
`sizeof...`	Queries the number of elements in a parameter pack.
C-style variadic functions	
Preprocessor macros	Can be variadic as well