Generic Programming

Compiled by
M S Anand

Department of Computer Science

Generic Programming
Variadic templates

Need for variadic templates

As far as C is concerned, the most commonly used library functions with variable number of arguments are:

```
int printf(const char *restrict format, ...);
int fprintf(FILE *restrict stream, const char *restrict format, ...);
int scanf(const char *restrict format, ...);
int fscanf(FILE *restrict stream, const char *restrict format, ...);
```

We can also write user defined functions which take a variable number of arguments:

An example is vnumarg.cpp

This implementation is specific for values of the **int** type. However, it is possible to write a similar function that is a function template

Generic Programming
Variadic templates

Rewriting the same program using function templates
vnumargs_template.cpp
Writing code like this, whether generic or not, has several important drawbacks:

- It requires the use of several macros: va_list (which provides access to the information needed by others), va_start (starts the iterating of the arguments), va_arg (provides access to the next argument), and va_end (stops the iterating of the arguments).
- Evaluation happens at runtime, even though the number and the type of the arguments passed to the function are known at compile-time.
- Variadic functions implemented in this manner are not type-safe. The va_ macros perform low-memory manipulation and type-casts are done in va_arg at runtime. These could lead to runtime exceptions.
- These variadic functions require specifying in some way the number of variable arguments. In the implementation of the earlier min function, there is a first parameter that indicates the number of arguments. The printf-like functions take a formatting string from which the number of expected arguments is determined. The printf function, for example, evaluates and then ignores additional arguments (if more are supplied than the number specified in the formatting string) but has undefined behavior if fewer arguments are supplied.

Generic Programming
Variadic templates

In addition to all these things, only functions could be variadic, prior to C++11. However, there are classes that could also benefit from being able to have a variable number of data members.

Variadic templates help address all these issues.

They are evaluated at compile-time, are type-safe, do not require macros, do not require explicitly specifying the number of arguments, and we can write both variadic function templates and variadic class templates.

Moreover, we also have variadic variable templates and variadic alias templates.

Variadic function templates

Variadic function templates are template functions with a variable number of arguments. They borrow the use of the ellipsis (...) for specifying a pack of arguments, which can have different syntax depending on its nature.

An example that rewrites the previous min function:

The code: variadic1.cpp

Generic Programming
Variadic templates

What we have here are two overloads for the min function.

The first is a function template with two parameters that returns the smallest of the two arguments.

The second is a function template with a variable number of arguments that recursively calls itself with an expansion of the parameters pack.

Although variadic function template implementations look like using some sort of compile-time recursion mechanism (in this case the overload with two parameters acting as the end case), in fact, they're only relying on overloaded functions, instantiated from the template and the set of provided arguments.

Generic Programming

Variadic templates

- The ellipsis (...) is used in three different places, with different meanings, in the implementation of a variadic function template, as can be seen in our example:
  - To specify a pack of parameters in the template parameters list, as in typename... Args. This is called a **template parameter pack**. Template parameter packs can be defined for type templates, non-type templates, and template template parameters.
  - To specify a pack of parameters in the function parameters list, as in Args... args. This is called a **function parameter pack**.
  - To expand a pack in the body of a function, as in args…, seen in the call min(args…). This is called a **parameter pack expansion**. The result of such an expansion is a comma-separated list of zero or more values (or expressions). This topic will be covered in more detail later in the course.

Generic Programming
Variadic templates

From the call min(1, 5, 3, -4, 9), the compiler is instantiating a set of overloaded functions with 5, 4, 3, and 2 arguments. Conceptually, it is the same as having the following set of overloaded functions:

```
int min(int a, int b)
{
        return a < b ? a : b;
}
int min(int a, int b, int c)
{
        return min(a, min(b, c));
}
int min(int a, int b, int c, int d)
{
        return min(a, min(b, min(c, d)));
}
int min(int a, int b, int c, int d, int e)
{
        return min(a, min(b, min(c, min(d, e))));
}
```

Generic Programming

Variadic templates

As a result, min(1, 5, 3, -4, 9) expands to min(1, min(5, min(3, min(-4, 9)))). This can raise questions about the performance of variadic templates. In practice, however, the compilers perform a lot of optimizations, such as inlining as much as possible. The result is that, in practice, when optimizations are enabled, there will be no actual function calls.

Understanding the expansion of parameter packs is key to understanding variadic templates.

Generic Programming

Parameter packs

A template or function parameter pack can accept zero, one, or more arguments. The standard does not specify any upper limit for the number of arguments, but in practice, compilers may have some. What the standard does is recommend minimum values for these limits but it does not require any compliance on them. These limits are as follows:

- For a function parameter pack, the maximum number of arguments depends on the limit of arguments for a function call, which is recommended to be at least 256.
- For a template parameter pack, the maximum number of arguments depends on the limit of template parameters, which is recommended to be at least 1,024.

The number of arguments in a parameter pack can be retrieved at compile time with the sizeof… operator. This operator returns a constexpr value of the std::size_t type.

Generic Programming

Parameter packs

In the following example, the sizeof… operator is used to implement the end of the recursion pattern of the variadic function template sum with the help of a constexpr if statement. If the number of the arguments in the parameter pack is zero (meaning there is a single argument to the function) then we are processing the last argument, so we just return the value. Otherwise, we add the first argument to the sum of the remaining ones.

The implementation:

```
template <typename T, typename... Args>
T sum(T a, Args... args)
{
        if constexpr (sizeof...(args) == 0)
                return a;
        else
                return a + sum(args...);
}
```

Generic Programming

Parameter packs

- This is semantically equivalent, but more concise, than the following classical approach for the variadic function template implementation:

```
template <typename T>
T sum(T a)
{
        return a;
}

template <typename T, typename... Args>
T sum(T a, Args... args)
{
        return a + sum(args...);
}
```

Generic Programming

Parameter packs

Notice that sizeof…(args) (the function parameter pack) and sizeof…(Args) (the template parameter pack) return the same value. On the other hand, sizeof…(args) and sizeof(args)... are not the same thing.

The former is the sizeof operator used on the parameter pack args.

The latter is an expansion of the parameter pack args on the sizeof operator.

These are both shown in the following example:

```
template<typename... Ts>
constexpr auto get_type_sizes()
{
        return std::array<std::size_t,
        sizeof...(Ts)>{sizeof(Ts)...};
}
auto sizes = get_type_sizes<short, int, long, long long>();
```

Generic Programming

Parameter packs

In the snippet (previous slide), sizeof…(Ts) evaluates to 4 at compile-time, while sizeof(Ts)... is expanded to the following comma-separated pack of arguments: sizeof(short), sizeof(int), sizeof(long), sizeof(long long).

Conceptually, the preceding function template, get_type_sizes, is equivalent to the following function template with four template parameters:

```
template<typename T1, typename T2, typename T3, typename T4>
constexpr auto get_type_sizes()
{
        return std::array<std::size_t, 4>
        {
                sizeof(T1), sizeof(T2), sizeof(T3), sizeof(T4)
        };
}
```

Generic Programming

Parameter packs

- Typically, the parameter pack is the trailing parameter of a function or template. However, if the compiler can deduce the arguments, then a parameter pack can be followed by other parameters including more parameter packs.

An example:

```
template <typename... Ts, typename... Us>
constexpr auto multipacks(Ts... args1, Us... args2)
{
        std::cout << sizeof...(args1) << ',' << sizeof...(args2) << '\n';
}
```

Generic Programming

Parameter packs

- This function is supposed to take two sets of elements of possibly different types and do something with them. It can be invoked such as in the following examples:

```
multipacks<int>(1, 2, 3, 4, 5, 6);       // 1,5
multipacks<int, int, int>(1, 2, 3, 4, 5, 6);  // 3,3
multipacks<int, int, int, int>(1, 2, 3, 4, 5, 6);     // 4,2
multipacks<int, int, int, int, int, int>(1, 2, 3, 4, 5, 6);    // 6,0
```

For the first call, the args1 pack is specified at the function call (as in multipacks<int>) and contains 1, and args2 is deduced to be 2, 3, 4, 5, 6 from the function arguments. Similarly, for the second call, the two packs will have an equal number of arguments, more precisely 1, 2, 3 and 4, 5, 6. For the last call, the first pack contains all the elements, and the second pack is empty. In all these examples, all the elements are of the int type.

Generic Programming

Parameter packs

However, in the following examples, the two packs contain elements of different types:

multipacks<int, int>(1, 2, 4.0, 5.0, 6.0); // 2,3
multipacks<int, int, int>(1, 2, 3, 4.0, 5.0, 6.0); // 3,3

For the first call, the args1 pack will contain the integers 1, 2 and the args2 pack will be deduced to contain the double values 4.0, 5.0, 6.0.

For the second call, the args1 pack will be 1, 2, 3 and the args2 pack will contain 4.0, 5.0, 6.0.

Generic Programming

Parameter packs

However, if we change the function template multipacks a bit by requiring that the packs be of equal size, then only some of the calls shown earlier would still be possible.

```
template <typename... Ts, typename... Us>
constexpr auto multipacks(Ts... args1, Us... args2)
{
        static_assert( sizeof...(args1) == sizeof...(args2),
                        "Packs must be of equal sizes.");
}

multipacks<int>(1, 2, 3, 4, 5, 6); // error
multipacks<int, int, int>(1, 2, 3, 4, 5, 6); // OK
multipacks<int, int, int, int>(1, 2, 3, 4, 5, 6); // error
multipacks<int, int, int, int, int, int>(1, 2, 3, 4, 5, 6);   // error
multipacks<int, int>(1, 2, 4.0, 5.0, 6.0); // error
multipacks<int, int, int>(1, 2, 3, 4.0, 5.0, 6.0); // OK
```

Generic Programming

Parameter packs

In the snippet (previous slide), only the second and the sixth calls are valid. In these two cases, the two deduced packs have three elements each.

In all the other cases, as resulting from the prior example, the packs have different sizes and the static_assert statement will generate an error at compile-time.

Generic Programming

Parameter packs

Multiple parameter packs are not specific to variadic function templates. They can also be used for variadic class templates in partial specialization, provided that the compiler can deduce the template arguments.

Consider the case of a class template that represents a pair of function pointers. The implementation should allow for storing pointers to any function.

To implement this, we define a primary template, func_pair, and a partial specialization with four template parameters:

- A type template parameter for the return type of the first function
- A template parameter pack for the parameter types of the first function
- A second type template parameter for the return type of the second function
- A second template parameter pack for the parameter types of the second function

Generic Programming

Parameter packs

The func_pair class template is:

```
template<typename, typename>
struct func_pair;

template<typename R1, typename... A1, typename R2, typename... A2>
struct func_pair<R1(A1...), R2(A2...)>
{
        std::function<R1(A1...)> f;
        std::function<R2(A2...)> g;
};
```

To demonstrate the use of this class template, let's also consider the following two functions:

```
bool twice_as(int a, int b)
{
        return a >= b*2;
}
double sum_and_div(int a, int b, double c)
{
        return (a + b) / c;
}
```

Generic Programming

Parameter packs

- We can instantiate the func_pair class template and use it to call these two functions as shown in the following snippet:

```
func_pair<bool(int, int), double(int, int, double)> funcs{
twice_as, sum_and_div };
funcs.f(42, 12);
funcs.g(42, 12, 10.0);
```

Parameter packs can be expanded in a variety of contexts.

Parameter packs

A *parameter pack* can be a type of parameter for templates.

Unlike previous parameters, which can only bind to a single argument, a parameter pack can pack multiple parameters into a single parameter by placing an ellipsis to the left of the parameter name.

In the template definition, a parameter pack is treated as a single parameter. In the template instantiation, a parameter pack is expanded and the correct number of the parameters are created.

According to the context where a parameter pack is used, the parameter pack can be either a *template parameter pack* or a *function parameter pack*.

Generic Programming

Parameter packs

Template parameter packs

A template parameter pack is a template parameter that represents any number (including zero) of template parameters. Syntactically, a template parameter pack is a template parameter specified with an ellipsis.

An example.

```
template<class...A> struct container{};
template<class...B> void func();
```

In this example, A and B are template parameter packs. According to the type of the parameters contained in a template parameter pack, there are three kinds of template parameter packs:

- Type parameter packs
- Non-type parameter packs
- Template template parameter packs

Generic Programming

Parameter packs

A type parameter pack represents zero or more type template parameters. Similarly, a non-type parameter pack represents zero or more non-type template parameters.

The following example shows a type parameter pack:

```
template<class...T> class X{};

X<> a;                    // the parameter list is empty
X<int> b;                 // the parameter list has one item
X<int, char, float> c;    // the parameter list has three items
```

In the above example, the type parameter pack T is expanded into a list of zero or more type template parameters.

Generic Programming

Parameter packs

- The following example shows a non-type parameter pack:

```
template<bool...A> class X{};

X<> a;                    // the parameter list is empty
X<true> b;                  // the parameter list has one item
X<true, false, true> c;   // the parameter list has three items
```

In this example, the non-type parameter pack A is expanded into a list of zero or more non-type template parameters.

Generic Programming

Parameter packs

In a context where template arguments can be deduced; for example, function templates and class template partial specializations, a template parameter pack does not need to be the last template parameter of a template. In this case, you can declare more than one template parameter pack in the template parameter list. However, if template arguments cannot be deduced, you can declare at most one template parameter pack in the template parameter list, and the template parameter pack must be the last template parameter.

Consider the following example:

```
// error
template<class...A, class...B>struct container1{};

// error
template<class...A,class B>struct container2{};
```

In this example, the compiler issues two error messages. One error message is for class template container1 because container1 has two template parameter packs A and B that cannot be deduced. The other error message is for class template container2 because template parameter pack A is not the last template parameter of container2, and A cannot be deduced.

Generic Programming

Parameter packs

Default arguments cannot be used for a template parameter pack.
Consider the following example:

```
template<typename...T=int> struct foo1{};
```

In this example, the compiler issues an error message because the template parameter pack T is given a default argument int.

Generic Programming

Parameter packs

Function parameter packs

A function parameter pack is a function parameter that represents zero or more function parameters. Syntactically, a function parameter pack is a function parameter specified with an ellipsis.

In the definition of a function template, a function parameter pack uses a template parameter pack in the function parameters. The template parameter pack is expanded by the function parameter pack. Consider the following example:

```
template<class...A> void func(A...args)
```

In this example, A is a template parameter pack, and args is a function parameter pack. You can call the function with any number (including zero) of arguments:

```
func();                 // void func();
func(1);                // void func(int);
func(1,2,3,4,5);        // void func(int,int,int,int,int);
func(1,'x', aWidget);   // void func(int,char,widget);
```

Generic Programming

Parameter packs

A function parameter pack is a *trailing function parameter pack* if it is the last function parameter of a function template. Otherwise, it is a *non-trailing function parameter pack*.

A function template can have trailing and non-trailing function parameter packs.

A non-trailing function parameter pack can be deduced only from the explicitly specified arguments when the function template is called.

If the function template is called without explicit arguments, the non-trailing function parameter pack must be empty, as shown in the following example:
ftemplate_pack1.cpp

Generic Programming

Parameter packs

In this example, function template func has two function parameter packs arg1 and arg2. arg1 is a non-trailing function parameter pack, and arg2 is a trailing function parameter pack.

When func is called with three explicitly specified arguments as func<int,int,int>(1,2,3,3,5,1,2,3,4,5), both arg1 and arg2 are deduced successfully.

When func is called without explicitly specified arguments as func(0,5,1,2,3,4,5), arg2 is deduced successfully and arg1 is empty.

In this example, the template parameter packs of function template func can be deduced, so func can have more than one template parameter pack.

Generic Programming

Parameter packs expansion

Pack expansion

A pack expansion is an expression that contains one or more parameter packs followed by an ellipsis to indicate that the parameter packs are expanded.

Consider the following example:

```
template<class...T> void func(T...a){};
template<class...U> void func1(U...b){
    func(b...);
}
```

In this example, T... and U... are the corresponding pack expansions of the template parameter packs T and U, and b... is the pack expansion of the function parameter pack b.

Generic Programming

Parameter packs expansion

A pack expansion can be used in the following contexts:

- Expression list
- Initializer list
- Base specifier list
- Member initializer list
- Template argument list
- Exception specification list

Expression List expansion

Sample program: expression_list.cpp

In this example, the switch statement shows the different positions of the pack expansion args... within the expression lists of the function func1. The output shows each call of the function func1 to indicate the expansion.

Initializer list

Sample program**:** initializer_list.cpp

In this example, the pack expansion args... is in the initializer list of the array res.

Generic Programming
Parameter packs expansion

Base specifier list

Sample program: base_specifier_list.cpp

In this example, the pack expansion baseC<A>... is in the base specifier list of the class template container. The pack expansion is expanded into four base classes baseC<a1>, baseC<a2>, baseC<a3>, and baseC<a4>.

The output shows that all the four base class templates are initialized before the instantiation of the class template container.

Generic Programming
Parameter packs expansion

Member initializer list

Sample program is: member_initializer_list.cpp

In this example, the pack expansion baseC<A>(12)... is in the member initializer list of the class template container. The constructor initializer list is expanded to include the call for each base class baseC<a1>(12), baseC<a2>(12), baseC<a3>(12), and baseC<a4>(12).

Generic Programming
Parameter packs expansion

- **Template argument list**

Sample program is: template_arg_list.cpp

In this example, the pack expansion C... is expanded in the context of template argument list for the class template container.

Exception specification list

Sample program is: e_spec_list.cpp

In this example, the pack expansion X... is expanded in the context of exception specification list for the function template func.

Generic Programming

Parameter packs expansion

If a parameter pack is declared, it must be expanded by a pack expansion. An appearance of a name of a parameter pack that is not expanded is incorrect. Consider the following example:

```
template<class...A> struct container;
template<class...B> struct container<B>{}
```

In this example, the compiler issues an error message because the template parameter pack B is not expanded.

Pack expansion cannot match a parameter that is not a parameter pack. Consider the following example:

```
template<class X> struct container{};

template<class A, class...B>
// Error, parameter A is not a parameter pack
void func1(container<A>...args){};

template<class A, class...B>
// Error, 1 is not a parameter pack
void func2(1...){};
```

Generic Programming

Parameter packs expansion

If more than one parameter pack is referenced in a pack expansion, each expansion must have the same number of arguments expanded from these parameter packs. Consider the following example:

```
struct a1{}; struct a2{}; struct a3{}; struct a4{}; struct a5{};

template<class...X> struct baseC{};
template<class...A1> struct container{};
template<class...A, class...B, class...C>
struct container<baseC<A,B,C...>...>:public baseC<A,B...,C>{};

int main(void){
    container<baseC<a1,a4,a5,a5,a5>, baseC<a2,a3,a5,a5,a5>,
            baseC<a3,a2,a5,a5,a5>,baseC<a4,a1,a5,a5,a5> > test;
    return 0;
}
```

Generic Programming

Parameter packs expansion

In this example, the template parameter packs A, B, and C are referenced in the same pack expansion baseC<A,B,C...>.... The compiler issues an error message to indicate that the lengths of these three template parameter packs are mismatched when expanding them during the template instantiation of the class template container.

Generic Programming

Parameter packs expansion

Partial specialization

Partial specialization is a fundamental part of the variadic templates feature. A basic partial specialization can be used to access the individual arguments of a parameter pack.

The following example shows how to use partial specialization for variadic templates:

```
// primary template
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<B,C...>{};
```

When the class template container is instantiated with a list of arguments, the partial specialization is matched in all cases where there are one or more arguments. In that case, the template parameter B holds the first parameter, and the pack expansion C... contains the rest of the argument list. In the case of an empty list, the partial specialization is not matched, so the instantiation matches the primary template.

Generic Programming

Parameter packs expansion

A pack expansion must be the last argument in the argument list for a partial specialization.

Consider the following example:

```
template<class...A> struct container;

// partial specialization
template<class B, class...C> struct container<C...,B>{};
```

In this example, the compiler issues an error message because the pack expansion C... is not the last argument in the argument list for the partial specialization.

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@pes.edu