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 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...);
}
```

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.

Generic Programming
Parameter packs

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.

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

Ạ 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

Generic Programming
Parameter packs expansion

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.

Generic Programming
Parameter packs 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.

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.

Generic Programming
Parameter packs expansion

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.

Generic Programming
Variadic Class templates

Class templates may have a variable number of template arguments. This is key to building some categories of types, such as tuple and variant, that are available in the standard library.

A simple implementation for a tuple class.
A tuple is a type that represents a fixed-size collection of heterogeneous values.

When implementing variadic function templates, we used a recursion pattern with two overloads, one for the general case and one for ending the recursion.

The same approach has to be taken with variadic class templates, except that we need to use specialization for this purpose.

Here is a minimal implementation for a tuple:
tuple_implementation.cpp

Generic Programming
Variadic Class templates

- The first class is the primary template. It has two template parameters: a type template and a parameter pack. This means, at the minimum, there must be one type specified for instantiating this template. The primary template tuple has two member variables: value, of the T type, and rest, of type tuple<Ts…>. This is an expansion of the rest of the template arguments.

  This means a tuple of N elements will contain the first element and another tuple; this second tuple, in turn, contains the second element and yet another tuple; this third nested tuple contains the rest.

  This pattern continues until we end up with a tuple with a single element. This is defined by the partial specialization tuple<T>. Unlike the primary template, this specialization does not aggregate another tuple object.

Generic Programming

Variadic Class templates

<u>How do we use this?</u>

```
tuple<int> one(42);
tuple<int, double> two(42, 42.0);
tuple<int, double, char> three(42, 42.0, 'a');
std::cout << one.value << '\n';
std::cout << two.value << ','
<< two.rest.value << '\n';
std::cout << three.value << ','
<< three.rest.value << ','
<< three.rest.rest.value << '\n’;
```

Although this works, accessing elements through the rest member, such as in three.rest.rest.value, is very cumbersome. And the more elements a tuple has, the more difficult it is to write code in this way.

<u>What is the way out?</u>

Use some helper function to simplify accessing the elements of a tuple.

Generic Programming

Variadic Class templates

The following is a snippet of how the previous could be transformed:

```
std::cout << get<0>(one) << '\n';
std::cout << get<0>(two) << ','
<< get<1>(two) << '\n';
std::cout << get<0>(three) << ','
<< get<1>(three) << ','
<< get<2>(three) << '\n’;
```

Here, get<N> is a variadic function template that takes a tuple as an argument and returns a reference to the element at the N index in the tuple.

How can the prototype for get<N> look like?

```
template <size_t N, typename... Ts>
typename nth_type<N, Ts...>::value_type & get(tuple<Ts...>& t);
```

Generic Programming
Variadic Class templates

The template arguments are the index and a parameter pack of the tuple types. Its implementation, however, requires some helper types. First, we need to know what the type of the element is at the N index in the tuple. This can be retrieved with the help of the following nth_type variadic class template:

```
template <size_t N, typename T, typename... Ts>
struct nth_type : nth_type<N - 1, Ts...>
{
        static_assert(N < sizeof...(Ts) + 1,
        "index out of bounds");
};

template <typename T, typename... Ts>
struct nth_type<0, T, Ts...>
{
        using value_type = T;
};
```

Note:
Static assertions are a way to check if a condition is true when the code is compiled. If it isn't, the compiler is required to issue an error message and stop the compiling process. The condition that needs to be checked is a constant expression.

Generic Programming

Variadic Class templates

Again, we have a primary template that uses recursive inheritance, and the specialization for the index 0.

The specialization defines an alias called value_type for the first type template (which is the head of the list of template arguments).

This type is only used as a mechanism for <u>determining the type of a tuple element</u>.

We need another variadic class template for retrieving the value. This is shown in the following listing:

VC1.cpp

Generic Programming

Variadic Class templates

We can see here the same recursive pattern, with a primary template and an explicit specialization.
The class template is called getter and has a single template parameter, which is a non-type template parameter.
This represents the index of the tuple element we want to access.
This class template has a static member function called get.
This is a variadic function template.
The implementation in the primary template calls the get function with the rest member of the tuple as an argument.
On the other hand, the implementation of the explicit specialization returns the reference to the member value of the tuple.

Generic Programming

Variadic Class templates

An actual implementation for the helper variadic function template **get**.

This implementation relies on the getter class template and calls its get variadic function template:

```
template <size_t N, typename... Ts>
typename nth_type<N, Ts...>::value_type &
get(tuple<Ts...>& t)
{
		return getter<N>::get(t);
}
```

Generic Programming
Fold expressions

A **fold expression** is an expression involving a parameter pack that folds (or reduces) the elements of the parameter pack over a binary operator.

The earlier implementation of sum that returned the sum of all its supplied arguments looked as follows:

```
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
Fold expressions

With fold expressions, this implementation that requires two overloads can be reduced to the following form:

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

There is no need for overloaded functions anymore. The expression (... + args) represents the fold expression, which upon evaluation becomes ((((arg0 + arg1) + arg2) + …) + argN). The enclosing parentheses are part of the fold expression.

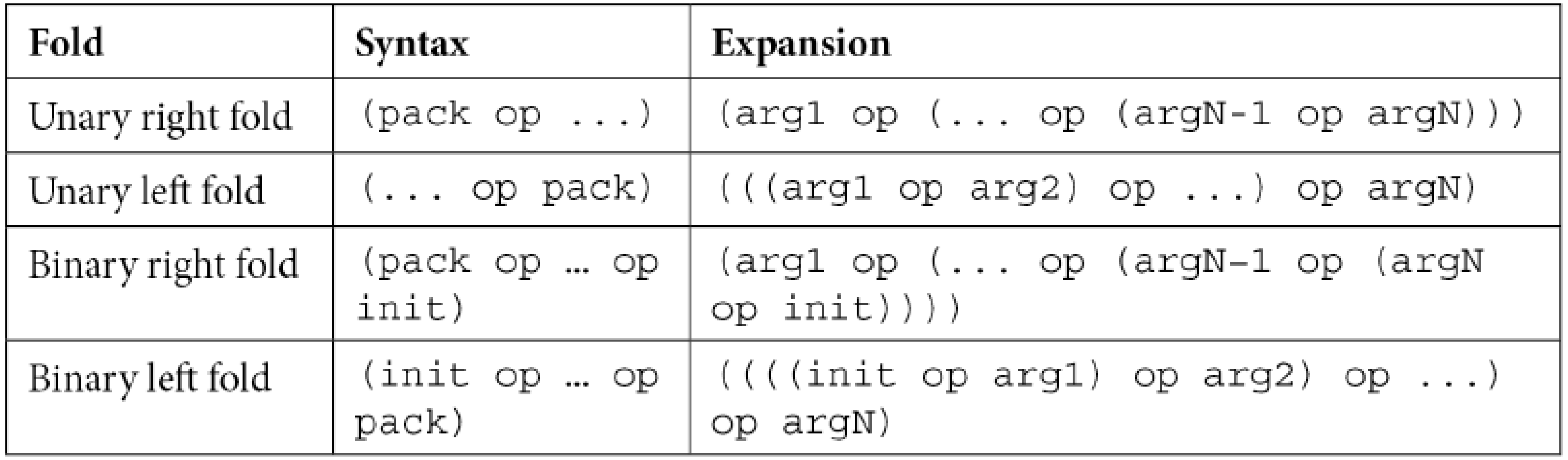
We can use this new implementation, just as we would use the initial one, as follows:

Generic Programming

Fold expressions

```
int main()
{
		std::cout << sum(1) << '\n’;
		std::cout << sum(1,2) << '\n’;
		std::cout << sum(1,2,3,4,5) << '\n';
}
```

There are four different types of folds, which are listed as follows:



Fold	Syntax	Expansion
Unary right fold	`(pack op ...)`	`(arg1 op (... op (argN-1 op argN)))`
Unary left fold	`(... op pack)`	`(((arg1 op arg2) op ...) op argN)`
Binary right fold	`(pack op … op init)`	`(arg1 op (... op (argN-1 op (argN op init))))`
Binary left fold	`(init op … op pack)`	`((((init op arg1) op arg2) op ...) op argN)`

Generic Programming
Fold expressions

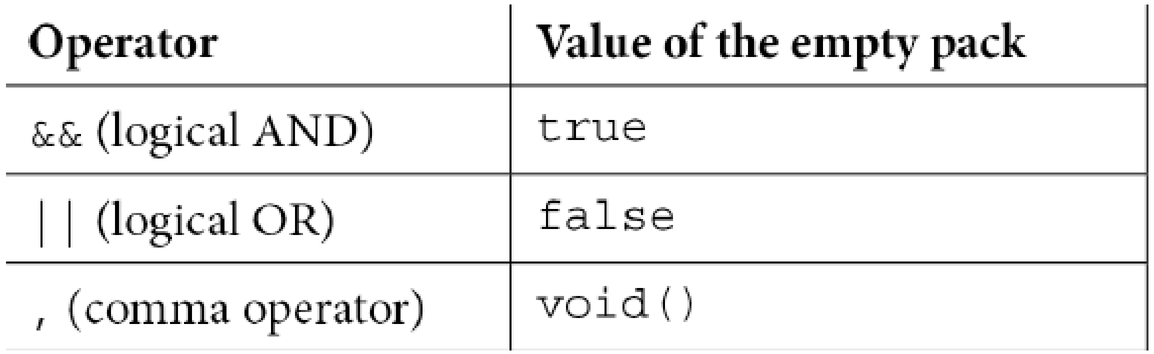
In this table, the following names are used:

• pack is an expression that contains an unexpanded parameter pack, and arg1, arg2, argN-1, and argN are the arguments contained in this pack.

• op is one of the following binary operators: + - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && || , .* ->*.

• init is an expression that does not contain an unexpanded parameter pack.

In a unary fold, if the pack does not contain any elements, only some operators are allowed. These are listed in the following table, along with the value of the empty pack:



Operator	Value of the empty pack
`&&` (logical AND)	`true`
`||` (logical OR)	`false`
`,` (comma operator)	`void()`

Generic Programming
Fold expressions

Unary and binary folds differ in the use of an initialization value, that is present only for binary folds. Binary folds have the binary operator repeated twice (it must be the same operator).
We can transform the variadic function template sum from using a unary right fold expression into one using a binary right fold by including an initialization value. Here is an example:

```
template <typename... T>
int sum_from_zero(T... args)
{
		return (0 + ... + args);
}
```

What is the difference between the sum and sum_from_zero function templates.

```
int s1 = sum(); // error
int s2 = sum_from_zero(); // OK
```

Generic Programming
Fold expressions

Calling sum without arguments will produce a compiler error, because unary fold expressions (over the operator + in this case) must have non-empty expansions. However, binary fold expressions do not have this problem, so calling sum_from_zero without arguments works and the function will return 0.

In these two examples with sum and sum_from_zero, the parameter pack args appears directly within the fold expression. However, it can be part of an expression, as long as it is not expanded. This is shown in the following example:

```
template <typename... T>
void printl(T... args)
{
		(..., (std::cout << args)) << '\n';
}

template <typename... T>
void printr(T... args)
{
		((std::cout << args), ...) << '\n';
}
```

Generic Programming
Fold expressions

Here, the parameter pack args is part of the (std::cout << args) expression. **This is not a fold expression**.
A fold expression is ((std::cout << args), ...). This is a unary left fold over the comma operator.

The printl and printr functions can be used as in the following snippet:
printl('d', 'o', 'g'); // dog
printr('d', 'o', 'g'); // dog

In both these cases, the text printed to the console is dog.
This is because the unary left fold expands to (((std::cout << 'd'), std::cout << 'o'), << std::cout << 'g') and the unary right fold expands to (std::cout << 'd', (std::cout << 'o', (std::cout << 'g'))) and these two are evaluated in the same way.
This is because a pair of expressions separated by a comma is evaluated left to right. This is true for the built-in comma operator. For types that overload the comma operator, the behavior depends on how the operator is overloaded.

Generic Programming

Fold expressions

Another example for using the parameter pack in an expression inside a fold expression.

The following variadic function template inserts multiple values to the end of a std::vector:

```
template<typename T, typename... Args>
void push_back_many(std::vector<T>& v, Args&&... args)
{
            (v.push_back(args), ...);
}
push_back_many(v, 1, 2, 3, 4, 5); // v = {1, 2, 3, 4, 5}
```

The parameter pack args is used with the v.push_back(args) expression that is folded over the comma operator. The unary left fold expression is (v.push_ back(args), ...).

Some example programs:

[SimpleFold.cpp] [Print_fold.cpp] [push_back_fold.cpp]

Generic Programming
Fold expressions

Fold expressions have several benefits over the use of recursion to implement variadic templates. These benefits are as follows:

- ❑ Less and simpler code to write.
- ❑ Fewer template instantiations, which leads to faster compile times.
- ❑ Potentially faster code since multiple function calls are replaced with a single expression. However, this point may not be true in practice, at least not when optimizations are enabled. The compilers might optimize code by removing these function calls.

Generic Programming
Variadic alias templates

Everything that can be templatized can also be made variadic. An alias template is an alias (another name) for a family of types.

A variadic alias template is a name for a family of types with a variable number of template parameters.

An example:

```
template <typename T, typename... Args>
struct foo
{
};
template <typename... Args>
using int_foo = foo<int, Args...>;
```

Generic Programming

Variadic alias templates

- The class template foo is variadic and takes at least one type template argument.
  int_foo, on the other hand, is only a different name for a family of types instantiated from the foo type with int as the first type template arguments.

  These could be used as follows:
  foo<double, char, int> f1;
  foo<int, char, double> f2;
  int_foo<char, double> f3;

  In this snippet, f1 on one hand and f2 and f3 on the other are instances of different foo types, as they are instantiated from different sets of template arguments for foo. However, f2 and f3 are instances of the same type, foo<int, char, double>, since int_foo<char, double> is just an alias for this type.

Generic Programming
Variadic alias templates

Another example

The standard library contains a class template called std::integer_sequence, which represents a compile-time sequence of integers, along with a bunch of alias templates to help create various kinds of such integer sequences. Although the code shown here is a simplified snippet, their implementation can, at least conceptually, be as follows:

vat1.cpp

There are three alias templates here:

- index_sequence, which creates an integer_sequence for the size_t type; this is a variadic alias template.
- index_sequence_for, which creates an integer_sequence from a parameter pack; this is also a variadic alias template.
- make_index_sequence, which creates an integer_sequence for the size_t type with the values 0, 1, 2, …, *N-1*. Unlike the previous ones, this is not an alias for a variadic template.

Generic Programming

Variadic variable templates

Variable templates may also be variadic. However, variables cannot be defined recursively, nor can they be specialized like class templates. Fold expressions, which simplify generating expressions from a variable number of arguments, are very handy for creating variadic variable templates.

Here is a variadic variable template called Sum that is initialized at compile-time with the sum of all integers supplied as non-type template arguments:

```
template <int... R>
constexpr int Sum = (... + R);
int main()
{
        std::cout << Sum<1> << '\n’;
        std::cout << Sum<1,2> << '\n’;
        std::cout << Sum<1,2,3,4,5> << '\n';
}
```

Generic Programming

Variadic variable templates

- This is similar to the sum function written with the help of fold expressions. However, in that case, the numbers to add were provided as function arguments. Here, they are provided as template arguments to the variable template. The difference is mostly syntactic; with optimizations enabled, the end result is likely the same in terms of generated assembly code, and therefore performance.

Variadic variable templates follow the same patterns as all the other kinds of templates although they are not used as much as the others.

Generic Programming

Name binding and dependent names

·The term name binding refers to the process of finding the declaration of each name that is used within a template. There are two kinds of names used within a template: **dependent names** and **non-dependent names**.

Dependent names are names that depend on the type or value of a template parameter that can be a type, non-type, or template parameter.
Names that don’t depend on template parameters are called non-dependent.
The name lookup is performed differently for dependent and non-dependent names:

- For dependent names, it is performed at the point of template instantiation.
- For non-dependent names, it is performed at the point of the template definition.

Generic Programming

Name binding and dependent names

An example:

namebinding.cpp

There are several points of reference that are marked in the comments on the right side.
At point [1], we have the declaration of a class template called parser.
This is followed at point [2] by the definition of a function called handle that takes a double as its argument.
The definition of the class template follows at point [3]. This class contains a single method called run that invokes a function called handle with the value 42 as its argument, at point [4].

Generic Programming

Name binding and dependent names

The name handle is a non-dependent name because it does not depend on any template parameter. Therefore, name lookup and binding are performed at this point. handle must be a function known at point [3] and the function defined at [2] is the only match.

After the class template definition, at point [5] we have the definition of an overload for the function handle, which takes an integer as its argument. This is a better match for handle(42), but it comes after the name binding has been performed, and therefore it will be ignored.

In the main function, at point [6], we have an instantiation of the parser class template for the type int. Upon calling the run function, the text processing a double: 42 will be printed to the console output.

Generic Programming
Name binding and dependent names

Another example:
namebinding2.cpp

This example is slightly different from the previous one. The parser class template is very similar, but the handle functions have become members of another class template.

Generic Programming

Name binding and dependent names

- At the point mark with [1] in the comments, we have the definition of a class template called handler. This contains a single, public method called handle that takes an argument of the T type and prints its value to the console.
- Next, at point [2], we have the definition of the class template called parser. This is similar to the previous one, except for one key aspect: at point [3], it invokes a method called handle on its argument.
- Because the type of the argument is the template parameter T, it makes handle a dependent name. Dependent names are looked up at the point of template instantiation, so handle is not bound at this point.
- At point [4], there is a template specialization for the handler class template for the type int. As a specialization, this is a better match for the dependent name. Therefore, when the template instantiation happens at point [6], handler<int>::handle is the name that is bound to the dependent name used at [3].
- Running this program will print handler<int>: 42 to the console.

Generic Programming

Name binding and dependent names

How is this topic related to templates?

Name lookup

Two-phase name lookup

As already explained, name lookup happens differently for dependent names and non-dependent names.

When the compiler passes through the definition of a template it needs to figure out whether a name is dependent or non-dependent.

Further name lookup depends on this categorization and happens either at the template definition point (for non-dependent names) or the template instantiation point (for dependent names).

Generic Programming
Name binding and dependent names

Therefore, instantiation of a template happens in two phases:

- The first phase occurs at the point of the definition when the template syntax is checked and names are categorized as dependent or non-dependent.

- The second phase occurs at the point of instantiation when the template arguments are substituted for the template parameters. Name binding for dependent names happens at this point.

This process in two steps is called **two-phase name lookup**.

Name binding and dependent names

Another example

two_phase.cpp

- In this snippet, we have two class templates: base_parser, which contains a public method called init, and parser, which derives from base_parser and contains a public method called parse.
- The parse member function calls a function called init and the intention is that it's the base-class method init that is invoked here.
- However, the compiler will issue an error, because it's not able to find init. The reason this happens is that init is a non-dependent name.
- Therefore, it must be known at the point of the definition of the parser template. Although a base_parser<T>::init exists, the compiler cannot assume it's what we want to call because the primary template base_parser can be later specialized and init can be defined as something else (such as a type, or a variable, or another function, or it may be missing entirely). Therefore, name lookup does not happen in the base class, only on its enclosing scope, and there is no function called init in parser.

Generic Programming
Name binding and dependent names

How do we solve this?
Make init a dependent name (How ?)

This can be done either by prefixing with this-> or with base_parser<T>::.

By turning init into a dependent name, its name binding is moved from the point of template definition to the point of template instantiation.

two_phase2.cpp

Suppose a specialization of base_parser for the int type is made available after the definition of the parser class template.

two_phase3.cpp

Generic Programming

Name binding and dependent names

When you run the program, the following text will be printed to the console:

```
specialized init
parse
init
parse
```

The reason for this behavior is that p1 is an instance of parser<int> and there is a specialization of its base class, base_parser<int> that implements the init function and prints “specialized init” to the console. On the other hand, p2 is an instance of parser<double>. Since a specialization of base_parser for the double type is not available, the init function from the primary template is being called and this only prints “init” to the console.

Name binding and dependent names

Dependent type names

So far we have seen examples where the dependent name was a function or a member function. However, there are cases when a dependent name is a type.

Example:

dependent_type1.cpp

In this snippet, base_parser is a class template that defines a type alias for T called value_type. The parser class template, which derives from base_parser, needs to use this type within its parse method. However, both value_type and base_parser<T>::value_type do not work, and the compiler issues an error.

Why ?

Generic Programming

Name binding and dependent names

value_ type does not work because it’s a non-dependent name and therefore it will not be looked up in the base class, only in the enclosing scope.

base_parser<T>::value_ type does not work either because the compiler cannot assume this is actually a type. A specialization of base_parser may follow and value_type could be defined as something else other than a type.

Generic Programming

Name binding and dependent names

In order to fix this problem, we need to tell the compiler the name refers to a type. Otherwise, by default, the compiler assumes it's not a type. This is done with the typename keyword, at the point of definition, shown in the following code snippet:

dependent_type2.cpp

There are actually two exceptions to this rule:

- When specifying a base class
- When initializing class members

Dig into this aspect further

Generic Programming

Name binding and dependent names

Dependent names can be templates too.

Dependent template names
There are cases when the dependent name is a template, such as a function template or a class template. However, the default behavior of the compiler is to interpret the dependent name as a non-type, which leads to errors concerning the usage of the comparison operator <.

An example: [dependent_tname1.cpp]()

The init function in base_parser is also a template. The attempt to call it using the base_parser<T>::init<int>() syntax, as seen at point [1], results in a compiler error. Therefore, we must use the template keyword to tell the compiler the dependent name is a template. This is done as shown at point [2].

The template keyword can only follow the scope resolution operator (::), member access through pointer (->), and the member access (.).
Examples of correct usage are X::template foo<T>(), this->template foo<T>(), and obj.template foo<T>().

Name binding and dependent names

·The dependent name does not have to be a function template. It can also be a class template:

An example: dependent_tname2.cpp

The token class is an inner class template of the base_parser class template. It can be either used as in the line marked with [1], where a type alias is defined (which is then used to instantiate an object) or as at line [2], where it is used directly to declare a variable.

Notice that the typename keyword is not necessary at [1], where the **using** declaration indicates we are dealing with a type, but is required at [2] because the compiler would otherwise assume it's a non-type name.

Generic Programming
Name binding and dependent names

Current instantiation
The requirement to use the typename and template keywords to disambiguate dependent names may be avoided in the context of a class template definition where the compiler is able to deduce some dependent names (such as the name of a nested class) to refer to the current instantiation.

This means some errors can be identified sooner, at the point of definition instead of the point of instantiation.

The following are the rules for considering a name as part of the current instantiation:

- An unqualified name (that does not appear on the right side of the scope resolution operator ::) found in the current instantiation or its non-dependent base
- A qualified name (that appears on the right side of the scope resolution operator ::) if its qualifier (the part that appears on the left side of the scope resolution operator) names the current instantiation and is found in the current instantiation or its non-dependent base
- A name used in a class member access expression where the object expression is the current instantiation and the name is found in the current instantiation or its non-dependent base

Note
It is said that a base class is a **dependent class** if it is a dependent type (depends on a template parameter) and is not in the current instantiation. Otherwise, a base class is said to be a **non-dependent class**.

Generic Programming
Name binding and dependent names

dependent_tname3.cpp

In the primary template parser, the names parser, parser<T>, and ::parser<T> all refer to the current instantiation.
However, parser<T*> does not.
The class token is a nested class of the primary template parser. In the scope of this class, token and parser<T>::token are both denoting the current instantiation. The same is not true for parser<T*>::token. This snippet also contains a partial specialization of the primary template for the pointer type T*. In the context of this partial specialization, parser<T*> is the current instantiation, but parser<T> is not.

Generic Programming
Template recursion

A simple implementation of factorial calculation is:

```
constexpr unsigned int factorial(unsigned int const n)
{
		return n > 1 ? n * factorial(n - 1) : 1;
}
```

A templatized implementation is:

[factorial_template.cpp]

Generic Programming
Template recursion

Wrt the above program, the first definition is the primary template.

It has a non-type template parameter representing the value whose factorial needs to be computed. This class contains a static constexpr data member called value, initialized with the result of multiplying the argument N and the value of the factorial class template instantiated with the decremented argument.

The recursion needs an end case and that is provided by the explicit specialization for the value 0 (of the non-type template argument), in which case the member value is initialized with 1.

Generic Programming

Template recursion

When encountering the instantiation factorial<4>::value in the main function, the compiler generates all the recursive instantiations from factorial<4> to factorial<0>.

Refer temp_recurse1.cpp

From these instantiations, the compiler is able to compute the value of the data member factorial<N>::value. It should be mentioned again that when optimizations are enabled, this code would not even be generated, but the resulting constant is used directly in the generated assembly code.

Generic Programming
Template recursion

·The implementation of the factorial class template is relatively trivial, and the class template is basically only a wrapper over the static data member value.

We can actually avoid it altogether by using a variable template instead. This can be defined as follows:

```
template <unsigned int N>
inline constexpr unsigned int factorial = N * factorial<N - 1>;
template <>
inline constexpr unsigned int factorial<0> = 1;
int main()
{
        std::cout << factorial<4> << '\n';
}
```

Generic Programming

Template recursion

- There is a striking similarity between the implementation of the factorial class template and the factorial variable template. For the latter, we have basically taken out the data member value and called it factorial. On the other hand, this may also be more convenient to use because it does not require accessing the data member value as in factorial<4>::value

Generic Programming

Template recursion

- There is a third approach for computing the factorial at compile time: using function templates. A possible implementation is shown next:

```
template <unsigned int n>
constexpr unsigned int factorial()
{
        return n * factorial<n - 1>();
}
template<> constexpr unsigned int factorial<1>()
{
        return 1;
}
template<> constexpr unsigned int factorial<0>()
{
        return 1;
}
int main()
{
        std::cout << factorial<4>() << '\n';
}
```

Generic Programming

Template recursion

Which of these three different approaches is the best is probably arguable. Nevertheless, the complexity of the recursive instantiations of the factorial templates remained the same. However, this depends on the nature of the template.

The following snippet shows an example of when complexity increases:

temp_recurse2.cpp

Generic Programming
Template recursion

There are two class templates in this example.

- The first is called wrapper and has an empty implementation (it doesn't actually matter what it contains) but it represents a wrapper class over some type (or more precisely a value of some type).
- The second template is called manyfold_wrapper. This represents a wrapper over a wrapper over a type many times over, hence the name manyfold_wrapper.

There is no end case for an upper limit of this number of wrappings, but there is a start case for the lower limit.

The full specialization for value 0 defines a member type called value_type for the unsigned int type.

As a result, manyfold_wrapper<1> defines a member type called value_type for wrapper<unsigned int>, manyfold_wrapper<2> defines a member type called value_type for wrapper<wrapper<unsigned int>>, and so on.

Generic Programming
Template recursion

- Therefore, executing the main function will print the following to the console:

```
unsigned int
struct wrapper<unsigned int>
struct wrapper<struct wrapper<unsigned int> >
struct wrapper<struct wrapper<struct wrapper<unsigned int> > >
```

Generic Programming
Template recursion

- The C++ standard <u>does not specify a limit for the recursively nested template instantiations but does recommend a minimum limit of 1,024</u>. However, this is only a recommendation and not a requirement. Therefore, different compilers have implemented different limits.

Recursive templates help us solve some problems in a recursive manner at compile time. Whether you use recursive function templates, variable templates, or class templates depends on the problem you are trying to solve or perhaps your preference. However, you should keep in mind there are limits to the depth template recursion works. Nevertheless, use template recursion judiciously.

Generic Programming

Template argument deduction

Explicitly specifying template arguments on every call to a function template (e.g., concat<std::string, int>(s, 3)) can quickly lead to unwieldy code. Fortunately, a C++ compiler can often automatically determine the intended template arguments using a powerful process called template argument deduction.

Although template argument deduction was first developed to ease the invocation of function templates, it has since been broadened to apply to several other uses, including determining the types of variables from their initializers.

Generic Programming
Template argument deduction

When you call a template function, you may omit any template argument that the compiler can determine or *deduce* by the usage and context of that template function call.

The compiler tries to deduce a template argument by comparing the type of the corresponding template parameter with the type of the argument used in the function call.

The two types that the compiler compares (the template parameter and the argument used in the function call) must be of a certain structure in order for template argument deduction to work.

The following lists these type structures:

ta_deduct.txt

Generic Programming
Template argument deduction

·T, U, and V represent a template type argument
10 represents any integer constant
i represents a template non-type argument
[i] represents an array bound of a reference or pointer type, or a non-major array bound of a normal array.
TT represents a template template argument
(T), (U), and (V) represents an argument list that has at least one template type argument
() represents an argument list that has no template arguments
<T> represents a template argument list that has at least one template type argument
<i> represents a template argument list that has at least one template non-type argument
<C> represents a template argument list that has no template arguments dependent on a template parameter

Generic Programming

Template argument deduction

- The following example demonstrates the use of each of these type structures. The example declares a template function using each of the above structures as an argument. These functions are then called (without template arguments) in order of declaration. The example outputs the same list of type structures:

template_deduce.cpp

Generic Programming
Template argument deduction

Deducing type template arguments

The compiler can deduce template arguments from a type composed of several of the listed type structures. The following example demonstrates template argument deduction for a type composed of several type structures:

```
template<class T> class Y { };

template<class T, int i> class X {
   public:
      Y<T> f(char[20][i]) { return x; };
      Y<T> x;
};

template<template<class> class T, class U, class V, class W, int i>
   void g( T<U> (V::*)(W[20][i]) ) { };

int main()
{
   Y<int> (X<int, 20>::*p)(char[20][20]) = &X<int, 20>::f;
   g(p);
}
```

Generic Programming

Template argument deduction

- The type Y<int> (X<int, 20>::*p)(char[20][20])T<U> (V::*)(W[20][i]) is based on the type structure T (U::*)(V):

T is Y<int>

U is X<int, 20>

V is char[20][20]

If you qualify a type with the class to which that type belongs, and that class (a nested name specifier) depends on a template parameter, the compiler will not deduce a template argument for that parameter. If a type contains a template argument that cannot be deduced for this reason, all template arguments in that type will not be deduced.

Generic Programming
Template argument deduction

```
template<class T, class U, class V>
  void h(typename Y<T>::template Z<U>, Y<T>, Y<V>) { };

int main() {
  Y<int>::Z<char> a;
  Y<int> b;
  Y<float> c;

  h<int, char, float>(a, b, c);
  h<int, char>(a, b, c);
  // h<int>(a, b, c);
}
```

Generic Programming

Template argument deduction

- The compiler will not deduce the template arguments T and U in typename Y<T>::template Z<U> (but it will deduce the T in Y<T>). The compiler would not allow the template function call h<int>(a, b, c) because U is not deduced by the compiler.

The compiler can deduce a function template argument from a pointer to function or pointer to member function argument given several overloaded function names. <u>However, none of the overloaded functions may be function templates, nor can more than one overloaded function match the required type</u>. The following example demonstrates this:

[deduce3.cpp]()

The compiler would not allow the call f(&g1) because g1() is a function template. The compiler would not allow the call f(&g2) because both functions named g2() match the type required by f().

Generic Programming
Template argument deduction

- The compiler <u>cannot deduce a template argument from the type of a default argument</u>. The following example demonstrates this:

```
template<class T> void f(T = 2, T = 3) { };

int main() {
   f(6);
//   f();
   f<int>();
}
```

Generic Programming

Template argument deduction

.The compiler allows the call f(6) because the compiler deduces the template argument (int) by the value of the function call's argument. The compiler would not allow the call f() because the compiler cannot deduce template argument from the default arguments of f().

The compiler cannot deduce a template type argument from the type of a non-type template argument. For example, the compiler will not allow the following:

```
template<class T, T i> void f(int[20][i]) { };

int main() {
   int a[20][30];
   f(a);
}
```

The compiler cannot deduce the type of template parameter T.

Generic Programming

Template argument deduction

If a template type parameter of a function template is a cv-unqualified rvalue reference, but the argument in the function call is an lvalue, the corresponding lvalue reference is used instead of the rvalue reference. However, if the template type parameter is a cv-qualified rvalue reference, and the argument in the function call is an lvalue, the template instantiation fails. For example:

```
template <class T> double func1(T&&);
template <class T> double func2(const T&&);

int var;

// The compiler calls func1<int&>(int&)
double a = func1(var);

// The compiler calls func1<int>(int&&)
double b = func1(1);

// error
double c = func2(var);

// The compiler calls func2<int>(const int&&)
double d = func2(1);
```

Generic Programming
Template argument deduction

In this example, the template type parameter of the function template func1 is a cv-unqualified rvalue reference, and the template type parameter of the function template func2 is a cv-qualified rvalue reference. In the initialization of variable a, the template argument var is an lvalue, so the lvalue reference type int& is used in the instantiation of the function template func1. In the initialization of variable b, the template argument 1 is an rvalue, so the rvalue reference type int&& remains in the template instantiation. In the initialization of c, the template type parameter T&& is cv-qualified, but var is an lvalue, so var cannot be bound to the rvalue reference T&&.

Generic Programming
Template argument deduction

Deducing non-type template arguments

The compiler cannot deduce the value of a major array bound unless the bound refers to a reference or pointer type. Major array bounds are not part of function parameter types. The following code demonstrates this:

```
template<int i> void f(int a[10][i]) { };
template<int i> void g(int a[i]) { };
template<int i> void h(int (&a)[i]) { };

int main () {
   int b[10][20];
   int c[10];
   f(b);
   // g(c);
   h(c);
}
```

Generic Programming
Template argument deduction

- The compiler would not allow the call g(c); the compiler cannot deduce template argument i.

The compiler cannot deduce the value of a non-type template argument used in an expression in the template function's parameter list. The following example demonstrates this:

```
template<int i> class X { };

template<int i> void f(X<i - 1>) { };

int main () {
  X<0> a;
  f<1>(a);
  // f(a);
}
```

Template argument deduction

·To call function f() with object a, the function must accept an argument of type X<0>. However, the compiler cannot deduce that the template argument i must be equal to 1 in order for the function template argument type X<i - 1> to be equivalent to X<0>. Therefore the compiler would not allow the function call f(a).
If you want the compiler to deduce a non-type template argument, the type of the parameter must match exactly the type of value used in the function call. For example, the compiler will not allow the following:

```
template<int i> class A { };
template<short d> void f(A<d>) { };

int main() {
   A<1> a;
   f(a);
}
```

Generic Programming
Template argument deduction

- The compiler will not convert int to short when the example calls f(). However, deduced array bounds may be of any integral type.

Template argument deduction also applies to the variadic templates feature.

Generic Programming
Forwarding references

The perfect forwarding problem
Let func(E1, E2, ..., En) be an arbitrary function call with generic parameters E1, E2, ..., En. We'd like to write a function wrapper such that wrapper(E1, E2, ..., En) is equivalent to func(E1, E2, ..., En). In other words, we'd like to define a function with generic parameters that forwards its parameters perfectly to some other function.

How it could have been done pre-C++11?
Let us assume all we need to forward is two arguments.

```
template <typename T1, typename T2>
void wrapper(T1 e1, T2 e2) {
    func(e1, e2);
}
```

This will obviously not work if func accepts its parameters by reference, since wrapper introduces a by-value passing step. If func modifies its by-reference parameter, it won't be visible in the caller of wrapper (only the copy created by wrapper itself will be affected).

Generic Programming

Forwarding references

We can make wrapper accept its parameters by reference. This should not interfere with func's taking parameters by value, because the call to func within wrapper will create the required copy.

```
template <typename T1, typename T2>
void wrapper(T1& e1, T2& e2) {
    func(e1, e2);
}
```

This has another problem;. rvalues cannot be bound to function parameters that are references, so the following completely reasonable calls will now fail:

```
wrapper(42, 3.14f);       // error: invalid initialization of non-const reference
                           //  from an rvalue

wrapper(i, foo_returning_float());   // same error
```

Making those reference parameters const won't cut it either, because func may legitimately want to accept non-const reference parameters.

Generic Programming
Forwarding references

What remains is the brute-force approach taken by some libraries: define overloads for both const and non-const references:

```
template <typename T1, typename T2>
void wrapper(T1& e1, T2& e2)                { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(const T1& e1, T2& e2)          { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(T1& e1, const T2& e2)          { func(e1, e2); }

template <typename T1, typename T2>
void wrapper(const T1& e1, const T2& e2)    { func(e1, e2); }
```

This is not scalable.
To make things worse, C++11 adds rvalue references to the mix (which we'd also want to forward correctly).

Generic Programming
Forwarding references

<u>Reference collapsing and special type deduction for rvalues</u>

```
template <typename T>
void baz(T t) {
  T& k = t;
}
```

What happens if we call this function as follows:

```
int ii = 4;
baz<int&>(ii);
```

In the template instantiation, T is explicitly set to int&. So what is the type of k inside? What the compiler "sees" is int& & - while this isn't something the user is allowed to write in code, <u>the compiler simply infers a single reference from this</u>.

With the addition of rvalue references in C++11, it became important to define what happens when various reference types augment (e.g. what does int&& & mean?).

Generic Programming
Forwarding references

·The result is the reference collapsing rule.
The rule is very simple. & always wins.
So & & is &, and so are && & and & &&. The only case where && emerges from collapsing is && &&. You can think of it as a logical-OR, with & being 1 and && being 0.

The other addition to C++11 relevant here is special type deduction rules for rvalue references in some cases. Given a function template like:

```
template <class T>
void func(T&& t) {
}
```

Here t is not an rvalue reference. When it appears in a type-deducing context, T&& acquires a special meaning. When func is instantiated, T depends on whether the argument passed to func is an lvalue or an rvalue. If it's an lvalue of type U, T is deduced to U&. If it's an rvalue, T is deduced to U.

Generic Programming
Forwarding references

```
func(4);            // 4 is an rvalue: T deduced to int

double d = 3.14;
func(d);            // d is an lvalue; T deduced to double&

float f() {...}
func(f());          // f() is an rvalue; T deduced to float

int bar(int i) {
  func(i);          // i is an lvalue; T deduced to int&
}
```

This rule may seem unusual and strange.

However, it starts making sense when we realize it was designed to solve the perfect forwarding problem.

Generic Programming
Forwarding references

Solving perfect forwarding with std::forward

Let's get back to our original wrapper template. Here's how it should be written in C++11:

```
template <typename T1, typename T2>
void wrapper(T1&& e1, T2&& e2) {
    func(forward<T1>(e1), forward<T2>(e2));
}
```

Note:

The *std::forward* function as the *std::move* function aims at implementing move semantics in C++. The function takes a forwarding reference. According to the *T* template parameter, std::forward identifies whether an lvalue or an rvalue reference has been passed to it and returns a corresponding kind of reference. *std::forward* helps to implement perfect forwarding.

Generic Programming
Forwarding references

Let's say we call:

```
int ii ...;
float ff ...;
```
wrapper(ii, ff);

Examining the first argument (since the second is handled similarly): ii is an lvalue, so T1 is deduced to int& following the special deduction rules. We get the call func(forward<int&>(e1), ...). Therefore, forward is instantiated with int& and we get this version of it:

```
int& && forward(int& t) noexcept {
    return static_cast<int& &&>(t);
}
```
Applying the reference collapsing rule:
```
int& forward(int& t) noexcept {
    return static_cast<int&>(t);
}
```
In other words, the argument is passed on by reference to func, as needed for lvalues.

Generic Programming
Forwarding references

The other case to handle is:

```
wrapper(42, 3.14f);
```

Here the arguments are rvalues, so T1 is deduced to int. We get the call func(forward<int>(e1), ...). Therefore, forward is instantiated with int and we get this version of it:

```
int&& forward(int&& t) noexcept {
    return static_cast<int&&>(t);
}
```

One can see forward as a pretty wrapper around static_cast<T&&>(t) when T can be deduced to either U& or U&&, depending on the kind of argument to the wrapper (lvalue or rvalue). Now we get wrapper as a single template that handles all kinds of forwarding cleanly.

The forward template exists in C++11, in the <utility> header, as std::forward.

Generic Programming
decltype specifier

- This specifier, introduced in C++11, returns the type of an expression.

It is usually used in templates together with the auto specifier.

Together, they can be used to declare the return type of a function template that depends on its template arguments, or the return type of a function that wraps another function and returns the result from executing the wrapped function

Generic Programming
decltype specifier

·The decltype specifier is not restricted for use in template code. It can be used with different expressions, and it yields different results based on the expression. The rules are as follows:

1. If the expression is an identifier or a class member access, then the result is the type of the entity that is named by the expression. If the entity does not exist, or it is a function that has an overload set (more than one function with the same name exists), then the compiler will generate an error.
2. If the expression is a function call or an overloaded operator function, then the result is the return type of the function. If the overloaded operator is wrapped in parentheses, these are ignored.
3. If the expression is an lvalue, then the result type is an lvalue reference to the type of expression.
4. If the expression is something else, then the result type is the type of the expression.

Generic Programming
decltype specifier

When the expression used with the decltype specifier contains a template, the template is instantiated before the expression is evaluated at compile time:

```
template <typename T>
struct wrapper
{
        T data;
};
decltype(wrapper<double>::data) e1; // double
int a = 42;
decltype(wrapper<char>::data, a) e2; // int&
```

The type of e1 is double, and wrapper<double> is instantiated for this to be deduced. On the other hand, the type of e2 is int& (as the variable a is an lvalue).

However, wrapper<char> is instantiated here even if the type is only deduced from the variable a (due to the use of the comma operator).

Generic Programming
decltype specifier

An example of a function template that returns the minimum of two values:

```
template <typename T>
T minimum(T&& a, T&& b)
{
		return a < b ? a : b;
}
```

We can use this as follows:

```
auto m1 = minimum(1, 5); // OK
auto m2 = minimum(18.49, 9.99);// OK
auto m3 = minimum(1, 9.99);  // error, arguments of different type
```

Generic Programming
decltype specifier

We could write a function template that takes two arguments of potentially different types and returns the minimum of the two.

This can look as follows:

```
template <typename T, typename U>
??? minimum(T&& a, U&& b)
{
        return a < b ? a : b;
}
```

What is the return type of this function?
This can be implemented differently, depending on the standard version you are using.

Generic Programming
decltype specifier

In C++11, we can use the auto specifier with a trailing return type, where we use the decltype specifier to deduce the return type from an expression.

```
template <typename T, typename U>
auto minimum(T&& a, U&& b) -> decltype(a < b ? a : b)
{
        return a < b ? a : b;
}
```

This syntax can be simplified if you're using C++14 or a newer version of the standard. The trailing return type is no longer necessary. You can write the same function as follows:

```
template <typename T, typename U>
decltype(auto) minimum(T&& a, U&& b)
{
        return a < b ? a : b;
}
```

Generic Programming
decltype specifier

It is possible to simplify this further and simply use auto for the return type, shown as follows:

```
template <typename T, typename U>
auto minimum(T&& a, U&& b)
{
        return a < b ? a : b;
}
```

Although decltype(auto) and auto have the same effect in this example, this is not always the case.

Generic Programming
decltype specifier

Consider the following example where we have a function returning a reference, and another function that calls it perfectly forwarding the argument:

```
template <typename T>
T const& func(T const& ref)
{
        return ref;
}

template <typename T>
auto func_caller(T&& ref)
{
        return func(std::forward<T>(ref));
}
```

Generic Programming
decltype specifier

```
int a = 42;
decltype(func(a)) r1 = func(a); // int const&
decltype(func_caller(a)) r2 = func_caller(a); // int
```

The function func returns a reference, and func_caller is supposed to do a perfect forwarding to this function. By using auto for the return type, it is inferred as int in the preceding snippet (see variable r2). In order to do a perfect forwarding of the return type, we must use decltype(auto) for it, as shown next:

```
template <typename T>
decltype(auto) func_caller(T&& ref)
{
        return func(std::forward<T>(ref));
}
int a = 42;
decltype(func(a)) r1 = func(a); // int const&
decltype(func_caller(a)) r2 = func_caller(a); // int const&
```

This time, the result is as intended, and the type of both r1 and r2 in this snippet is int const&.

Generic Programming
decltype specifier

In summary, decltype is a type specifier used to deduce the type of an expression. It can be used in different contexts, but its purpose is for templates to determine the return type of a function and to ensure the perfect forwarding of it.

Generic Programming
std::declval type operator

The std::declval type operator

The std::declval is a utility type operation function, available in the <utility> header. It’s in the same category as functions such as std::move and std::forward that we have already seen. What it does is very simple: it adds an rvalue reference to its type template argument. The declaration of this function looks as follows:

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

This function has no definition and therefore it cannot be called directly. It can only be used in **unevaluated contexts** – decltype, sizeof, typeid, and noexcept. These are compile-time-only contexts that are not evaluated during runtime. The purpose of std::declval is to aid with dependent type evaluation for types that do not have a default constructor or have one but it cannot be accessed because it’s private or protected.

Generic Programming

std::declval type operator

Consider a class template that does the composition of two values of different types, and we want to create a type alias for the result of applying the plus operator to two values of these types.

How do you define such a type alias?

Start with the following form:

```
template <typename T, typename U>

struct composition
{
        using result_type = decltype(???);
};
```

We can use the decltype specifier but we need to provide an expression. We cannot say decltype(T + U) because these are types, not values. We could invoke the default constructor and, therefore, use the expression decltype(T{} + U{}).

Generic Programming
std::declval type operator

This can work fine for built-in types such as int and double, as shown in the following snippet:

```
static_assert(
std::is_same_v<double,
composition<int, double>::result_type>);
```

It can also work for types that have an (accessible) default constructor. But it cannot work for types that don’t have a default constructor. The following type wrapper is such an example:

```
struct wrapper
{
		wrapper(int const v) : value(v){}
		int value;
		friend wrapper operator+(int const a, wrapper const& w)
		{
				return wrapper(a + w.value);
		}
		friend wrapper operator+(wrapper const& w, int const a)
		{
				return wrapper(a + w.value);
		}
};
```

Generic Programming
std::declval type operator

```
// error, no appropriate default constructor available

static_assert(
std::is_same_v<wrapper,
composition<int,wrapper>::result_type>);
```

The solution here is to use std::declval(). The implementation of the class template composition would change as follows:

```
template <typename T, typename U>
struct composition
{
        using result_type = decltype(std::declval<T>() +
        std::declval<U>());
};
```

With this change, both the static asserts previously shown compile without any error.

Generic Programming

std::declval type operator

- This function avoids the need to use particular values to determine the type of an expression. It produces a value of a type T without involving a default constructor.

The reason it returns an rvalue reference is to enable us to work with types that cannot be returned from a function, such as arrays and abstract types.

The definition of the wrapper class earlier contained two friend operators. Friendship, when templates are involved, has some particularities.

Generic Programming
General friendship

General Friendship
A class or a class template can grant friendship to each instance of a class template or a function template.

friendship.cpp

Line (1) and line (2) forward declare the function template myFriendFunction and the class template MyFriend. The function template myFriendFunction is defined in line (3), and the class template MyFriend in line (4). The classes GrantingFriendshipAsClass and GrantingFriendshipAsClassTemplate grant the function template myFriendFunction and the class template MyFriend friendship. Due to the friendship, both templates can directly invoke the private member "secrete" of the class and the class template.

Generic Programming

General friendship

- There is a pitfall involved in the class template GrantingFriendShipAsClassTemplate. Usually, you call the first type parameter of a template T. When you use – such as in the following code snippet – the same type parameter name for the class template and the function template myFriendFunction or the class template, MyFriend, an error occurs. The name T of myFriendFunction or MyFriend shadows the name T of the class template GrantingFriendshipAsClassTemplate.

The following code snippet displays the pitfall.

```
template <typename T>
class GrantingFriendshipAsClassTemplate{

  template <typename T> friend void myFriendFunction(T);
  template <typename T> friend class MyFriend;

  std::string secret{"Secret from GrantingFriendshipAsClassTemplate."};

}
```

Generic Programming

Special friendship

Special Friendship

A special friendship is a friendship that depends on the type of template parameter.

specialfriendship.cpp

The class GrantingFriendshipAsClass grants friendship to the full specialization of the function template myFriendFunction for int (line 1) and the class template MyFriend for int (line 2). The same holds for the class template GrantingFrandshipAsClassTemplate. Lines (3) is unique because it grants friendship to the full specialization for MyFriend having the same type parameter as the class template GrantingFrandshipAsClassTemplate. Consequently, the function template myFriendFunction can invoke the secret of the class GrantingFriendshipAsClass when myFriendFunctions is a full specialization for int (line 4) or GrantingFriendshipAsClassTemplate has the same type, such as myFriendFunction (line 5). The corresponding argumentation holds for the class template MyFriend (line 6).

Generic Programming
Friend to types

Friend to Types
A class template can also grant its friendship to a type parameter.

typefriendship.cpp

The class Bank grants friendship to its type parameter T. Consequently, an Account can access the secret of the bank instantiation for Account: Bank<Account> (line 1).

Generic Programming

Type traits

- WIP

Generic Programming
Type traits

- WIP

Generic Programming

Type traits

- WIP

Generic Programming
Type traits

- WIP

Generic Programming

Type traits

- WIP

Generic Programming
Type traits

WIP

PES UNIVERSITY ONLINE

THANK YOU

M S Anand
Department of Computer Science Engineering

anandms@pes.edu