Allowing multiple template parameter packs of different types

Document #:	D{to be assigned}R0
Date:	2022-11-16
Project:	Programming Language C++ Core Language Wording

Reply-to:	Anoop Rana <ranaanoop986@gmail.com>

Abstract

The aim of this document is to make program-2 well-formed so that the template argument matching process becomes more intuitive and consistent. Additionally in the process of making program-2 well-formed we also aim to make program-1 well-formed.

1 Introduction

Currently program-1 is ill-formed as per [temp.param#14] as `U` can neither be deduced from the parameter-type-list nor specified as template parameter packs are supposedly greedy. We say supposedly because currently the standard doesn't have a separate clause saying that template parameter packs consume every explicitly passed template argument except for a comment given in [temp.param#14].This is what [temp.param#14] currently says:

> A template parameter pack of a function template shall not be followed by another template parameter unless that template parameter can be deduced from the parameter-type-list ([dcl.fct]) of the function template or has a default argument ([temp.deduct]).

```
// U can be neither deduced from the parameter-type-list nor specified
template<class... T, class... U> void f() { } // error
```

Program 1: Ill-formed as per temp.param

We observe compiler divergence here(with the call `f<int, int>();`) as clang and gcc compiles this call while msvc rejects it. That is, two of three major compilers compiles the above program with the call `f<int,int>();` with `T={int, int}, U={ }`. Also since it is possible for a template parameter pack like `U` to be empty, we propose to make program-1 well-formed. This can be done by adding a separate clause in the standard saying something like a template parameter pack `T` consumes all the explicitly provided template argument(s) **of the same kind** until an argument of different kind is encountered so that the template parameter `U` **directly** following `T` will match with the remaining argument(s) with the same process as described above. That is, this process will repeat for each parameter(s). The qualifier "directly" in the previous sentence means that there is no parameter in between `T` and `U`. In case there are no more arguments left after matching explicitly provided argument(s) with a parameter, the remaining template parameter packs(if any) will be empty.

2 Problem with current wording

We also note that according to the current wording program-2 is also ill-formed. This doesn't make sense because a template "type" parameter pack can't possibly consume a template "non-

type" argument. In particular, the packs `T` and `U` are of different kinds, so that if a call like `f<int, int, true, false>();` is made in program-2 then the first two "type" arguments can only match with `T` while the last two "non-type" arguments can match only with `U`. Basically **a template parameter pack should consume(or be greedy) only for explicitly provided template arguments of the same kind**.

Next is given the program that is ill-formed as per the current wording.

```
//-------------------vvvv---------------------->note U is a non-type template parameter pack
template<class... T, bool... U> void f() { } // ill-formed as per current wording
```

Program 2: Ill-formed as per current wording

3 Motivation and Scope

It is already possible for a template parameter pack to be the empty sequence `{ }` so there is no reason to make program-1 ill-formed when it can be well-formed with `U` being `{ }`. Note again that two of the three majore compilers(gcc and clang) already compiles program-2 with call `f<int, int>();`.

4 Proposed-wording

4.1 Addition of clause

The first change is to add a separate clause in the standard which more or less have the same meaning as saying that a template parameter pack `T` consumes all the explicitly provided template argument(s) **of the same kind** until an argument of different kind is encountered so that the template parameter `U` **directly** following `T` will match with the remaining argument(s) with the same process as described above. That is, this process will repeat for each parameter(s). The qualifier "directly" in the previous sentence means that there is no parameter in between `T` and `U`. In case there are no more arguments left after matching explicitly provided argument(s) with a parameter, the remaining template parameter packs(if any) will be empty.

4.2 Change in temp.param

Second change that will be needed is in [temp.param#14] that will make program-2 well-formed. The first change(addition of separate clause) highly influence this second change. Maybe even the addition of separate clause can be done in [temp.param#14].

5 Before/After Comparisons

A comparison table is given below.

Template	Usage	GCC	CLANG	MSVC	Standard	Proposed	Proposed-T	Proposed-U	Proposed-V
template<class... T, class... U>void f();	f<int, int>();	✓	✓	Err	Err	✓	{int, int}	{}	
template<class... T, class U>void f();	f<int, int>();	Err	Err	Err	Err	Err			
template<class...T, bool... U>void f();	f<int, int, true, false>();	Err	Err	Err	Err	✓	{int, int}	{true, false}	
template<class...T, bool... U>void f();	f<int, int>();	✓	✓	Err	Err	✓	{int,int}	{}	
template<class...T, bool U>void f();	f<int, int>();	Err	Err	Err	Err	Err			
template<class...T, bool... U, class... V>void f();	f<int, int, true>();	Err	Err	Err	Err	✓	{int, int}	{true}	{}
template<class...T, bool... U, class... V>void f();	f<int, int, true, double>();	Err	Err	Err	Err	✓	{int, int}	{true}	{double}
template<class...T, bool... U, class... V>void f();	f<1, int>();	Err	Err	Err	Err	Err			
template<class...T, auto... U>void f();	f<int, int>();	✓	✓	Err	Err	✓	{int, int}	{}	
template<class...T, auto... U>void f();	f<int, int, true>();	Err	Err	Err	Err	✓	{int, int}	{true}	
template<auto... T, int... U>int f();	f<1>();	✓	✓	Err	Err	✓	{1}	{}	
template<int... T, auto... U>int f();	f<1>();	✓	✓	Err	Err	✓	{1}	{}	
template<class... T, template<class>class... U>void f();	f<int, int>();	✓	✓	Err	Err	✓	{int, int}	{}	
template<class... T, template<class>class... U>void f();	f<int, int, X>();	Err	Err	Err	Err	✓	{int, int}	X	

In the above table, `X` is a class template with a single template type parameter.

6 Impact on the standard

This proposal makes function templates involving multiple template parameter packs more useful. It will only allow some of the previously rejected programs(which have multpile template parameter packs) to be now well-formed and not the opposite. That is, some previously rejected program can now be compiled and no previously accepted program(involving multiple template parameter packs) will be rejected.