Allowing multiple template parameters packs of different types

Document #:	P{to be assigned}R1
Date:	2022-11-14
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 2 well-formed we also aim to clarify(by adding a separate clause for it) which of the two interpretations given in this document is correct.

1 Introduction

Currently program-1.1 is ill-formed as per [temp.param#14]. There are two interpretations of understanding this. Note that the standard doesn't have a separate clause supporting any of these interpretation except a comment given in [temp.param#14] supporting Interpretation of greedy packs. The proposed wording in this document applies equally well for both interpretation.

1.1 Interpretation of greedy packs

According to this interpretation, the template parameter packs `T` and `U` are greedy meaning that `T` will consume each of the explicitly provided template arguments so that there is no way for `U` to be known(as it can't be deduced nor it can be explicitly specified). This is what the comment in the example [temp.param#14] indicates: *U can neither be deduced from the parameter-type-list nor specified*. Also note that except for the above quoted comment there is no other separate clause saying that template packs consume every explicitly provided arguments. 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 1.1: Ill-formed as per temp.param

We also note that this interpretation doesn't explain why `U` can't be the empty sequence `U={}`. That is, if we take this interpretation as correct, then instead of the program being ill-formed(which it is per current wording), why `U` can't just be the empty sequence.

1.2 Interpretation of multiple possibilities

Another possible interpretation is that even if we explicitly provide template arguments there is no way to tell which arguments match which parameters. For example, say the user write `f<int, double, int, int>();` then some of the possible combinations for the sequences are `T = {int}, U = {double, int, int}` or `T = {int, double}, U = {int, int}` or `T = {}, U = {int, double, int, int}` and so on. Basically there are many options and no way of knowing which one to pick/use among them. So it makes sense to disallow program-1.1.

2 Problem with current wording

The above discussion shows that it makes sense to reject program-1.1. But what doesn't make sense is that when the underlying types of `T` and `U` are different, the program will still be ill-formed as per the current wording. In particular, program-2 where `T` is a template **type parameter** pack while `U` is a template **non-type parameter** pack is still ill-formed as per the current wording. The aim of this document is to make program-2 well-formed.

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

First let's consider why program-2 should be well-formed even if we take Interpretation of multiple possibilities as correct. In this case, now there isn't any kind of ambiguity while choosing the sequence for `T` and `U`. For example, say the user calls the function like `f<int, double, true, false>();`. Now the **only possibility** is that `T = {int, double}, U = {true, false}`. And as this is the only possibility for this call, it makes sense for 2 to be well-formed. All this is because the underlying types of `T` and `U` are different. One is a template type parameter pack and the latter is a template non-type parameter pack. And so the **explicitly passed arguments cannot possibly be confused with each other** as the former's sequence expects "types" as argument(s) while the latter's sequence expects "values" of type `bool` as argument(s).

Now let's consider why program-2 should be well-formed even if we take Interpretation of greedy packs as correct. Note that we're not saying that interpretation 1 is correct but instead that even if it is correct, then also program-2 should still be well-formed. In this case, consider again the call `f<int, double, true, false>();`. Now, since a template type parameter can only match

a template type argument while a template non-type parameter can only match a template non-type argument, it makes sense that `T={int, double},U={true, false}`. Note again that this is because `T` and `U` are of different types. One is a template type parameter pack while template non-type parameter pack. So even if greedy matching is the correct interpretation, it can't possibly match the last two arguments(`true, false`) to `T`.

4 Proposed-wording

4.1 Change in temp.param

Make the following changes in [temp.param#14]:

> A template parameter pack T of a function template shall not be followed by another template parameter U 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]) **or the following parameter U has a different underlying type than any of its preceeding parameters**.

4.2 Addition of clause

In addition to the above indicated change in [temp.param#14] we also propose to add a separate clause for clarifying which interpretation(among the two given in this document) is correct. As to our current understanding, we support Interpretation of multiple possibilities over interpretation 1 because interpretation 1 doesn't explain why `U` can't be the empty sequence.

Thus, a clause clarifying the same should be added.

5 Before/After Comparisons

5.1 Type Pack vs Non-Type Pack

Before	After

`template<typename... T, bool... U> void f(){}`
`f<int, double, true, false>(); //ill-formed`
	`template<typename... T, bool... U> void f(){}`
	`f<int, double, true, false>(); //well formed`
	`T = {int, double}, U = {true, false}`

5.2 Type pack vs Non-Type Non Pack

Before	After

`template<typename... T, bool U> void f(){}`
`f<int, double, true>(); //ill-formed`
	`template<typename... T, bool U> void f(){}`
	`f<int, double, true>(); //well formed`
	`T = {int, double}, U = {true}`

5.3 Type Pack vs Template Pack

Before

`template<typename... T, template<typename>typename... U> void f(){}`
`template<typename> struct C{};`
`f<int, double, C>(); //ill-formed`

After

`template<typename... T, template<typename>typename... U> void f(){}`
`template<typename> struct C{};`
`f<int, double, C>(); //well formed with {T = int, double}, U = {C}`

5.4 Type Pack vs Non-Type Non Pack Revisited

Before	After
`template<typename... T, bool U> void f(){}` `f<int, double>(); //ill-formed`	`template<typename... T, bool U> void f(){}` `f<int, double>(); //ill formed`

5.5 Type Pack vs Non-Type Pack Revisited

Before	After

`template<typename... T, bool... U> void f(){}`
`f<int, double>(); //ill-formed`
	`template<typename... T, bool... U> void f(){}`
	`f<int, double>(); //well formed`
	`T = {int, double}, U = { }`

5.6 Three parameter packs ill-formed

Before

```
template<class... Ts, bool... Bs, class... Us> void f(){} //ill-formed
```

After

```
template<class... Ts, bool... Bs, class... Us> void f(){} //ill-formed as Us and Ts are both
                                                          template "type" parameter packs
```

5.7 Three parameter packs well-formed

Before

```
template<class... Ts, bool... Bs, template<typename>typename... Us> void f(){} //ill-formed
```

After

```
template<class... Ts, bool... Bs, template<typename>typename... Us> void f(){} //well-formed
as all Ts, Bs, and Us are of different underlying types
```

6 Impact on the standard

This proposal makes function templates involving multiple template parameter packs more useful and requires minimal change in the core standard language. 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.