

JTC1/SC22/WG14 - N2779

Title: Consistency of Parameters Declared as Arrays

Author: Martin Uecker, University of Göttingen

Date: 2021-07-11

Prior Art: GCC

As discussed in N2074 and N2660, it would be helpful if function declarations that have parameters declared as arrays are required to be consistent. Some compilers and tools already diagnose such inconsistencies. A closer analysis of this question reveals that there are two orthogonal problems:

1. Parameters declared as arrays are not required to be compatible because they are adjusted to pointers (6.7.3p6).

```
void foo(double x[3]);           // decl 1
void foo(double x[4]);           // decl 2

void foo(double *x);             // decl 1 with adjusted parameter type
void foo(double *x);             // decl 2 with adjusted parameter type
```

2. In a function declarator that is not part of a function definition, the size expression of array types which are not integer constant expressions are replaced by `*` and the array is then declared as a variably-length array of unspecified size.

```
size_t m;
void f(size_t n, double x[n], double y[m], double (*z)[n]); // decl 3
void f(size_t n, double x[*], double y[*], double (*z)[*]); // after replacement
```

Note that this affects all parameters with VM types and not just VLAs. This is more difficult to fix, because the expression is only evaluated after the function is invoked, and the normal mechanism that makes it UB if size expressions do not evaluate to the same value at runtime is ineffective..

Both problems are orthogonal, with the first problem affecting all parameters that are arrays and the second problem affecting all parameters that have VM types. The following example is affected by both problems at the same time:

```
void foo(size_t n, double x[n]);
```

In this paper, we address only the first problem (and also ignore static). We propose to make the unadjusted type stay part of the function type and adjust only the types of the parameters themselves, i.e. inside the function body and for assignment of the argument to the parameter when performing the function call. We then propose to modify the rules for type compatibility and composite types for function parameters in the following way:

1. An array type is compatible to a pointer type when it is compatible after adjustment.
2. The composite type when one type is a pointer type and the other an array is the array type.

The following declarations declare functions with compatible type, while the preceding declarations 1 and 2 declare mutually incompatible function types. As usual, the composite type is the type with the most information:

```
int bar(double x[]);
int bar(double x[3]);
int bar(double *x);
int bar(double x[3]);           // declaration with composite type
```

Proposed wording (relative to N2596)

6.5.2.2 Function calls

Constraints

2 If the expression that denotes the called function has a type that includes a prototype, the number of arguments shall agree with the number of parameters. Each argument shall have a type such that its value may be assigned to an object with the unqualified version of the **adjusted** type of its corresponding parameter.

6.2.7 Compatible type and composite type

3 A composite type can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

--- If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters **according to the rules described in 6.7.6.3. If a parameter of array or function type is compatible after adjustment to a pointer type, the composite type is the array or function type, respectively.**

6.7.6.3 Function declarators

6 **To determine the type of a parameter itself but not for determining the function type that contains the declaration of the parameter in the parameter type list**, a declaration of a parameter as "array of type" **shall be** adjusted to "qualified pointer to type", where the type qualifiers (if any) are those specified within the [and] of the array type derivation.

7 **To determine the type of the parameter itself but not for determining the function type that includes the declaration of the parameter in the parameter type list**, a declaration of a parameter as "function returning type" **shall be** adjusted to "pointer to function returning type", as in 6.3.2.1

14 For two function types to be compatible, both shall specify compatible return types. Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator; corresponding parameters shall have compatible types. If one type has a parameter type list and the other type has none and is not part of a function definition, the parameter list shall not have an ellipsis terminator. In the determination of type compatibility and of a composite type, each parameter declared with function or array type is **taken as having considered to be compatible not only to a compatible function or array type, respectively, but also to the corresponding** adjusted type, and each parameter declared with qualified type is taken as having the unqualified version of its declared type.

21 EXAMPLE 6 The following function prototype declarators are incompatible:

```
void g(int x[5]);  
void g(int x[3]);
```

Acknowledgements: I want to thank Aaron Ballman and Robert Seacord for helpful comments.