JTC1/SC22/WG14 - N2906
Title: Consistency of Parameters Declared as Arrays (updates N2779)
Author: Martin Uecker
Date: 2021-12-29
Prior Art: GCC

Changes since N2779:

- Propose wording for main function
- Propose wording for typedef
- Add examples / footnote
- Properly construct composite type

Introduction

As discussed in N2074 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 (see N2779). In this paper, we address the problem that 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
```

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]);          // compatible 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 type is compatible after adjustment to a pointer type, the composite type is an array type with the same size as the array type and with an element type that is the composite type of the element type of the array type and the type referenced by the pointer type.**

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**~~**is** 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**~~**is** 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.[XX)]

21 EXAMPLE 6 The following function prototype declarators are not compatible:

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

22 EXAMPLE 7 The following function prototype declarators are compatible:

```
void g(int x[const 5]);
void g(int x[5]);                  // composite type
void g(int * restrict x);
void g(int x[*]);
void g(int x[]);
```

[xx] Type qualifiers within [and] of an array type derivation are also ignored.

Additional Change 1

5.1.2.2.1 Program startup

with two parameters ~~**(referred to here as argc and argv , though any names may be used, as they are local to the function in which they are declared)**~~:

int main(int argc, char *argv[]) { /* ... */ }

or ~~**equivalent;**~~ **using any other function definition of the identifier main that has a compatible type;**[10] or in some other implementation-defined manner.

Additional Change 2a

6.7 Declarations

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that

– a typedef name may be ~~**redefined to denote the same type as it currently does, provided that type is not a variably modified type;**~~ **redeclared as specified in 6.7.8;**

6.7.8 Type definitions

Constraints

3 If a typedef name is redefined in the same scope the type specified shall be compatible to the previous type.

Semantics

4 When a typedef name is redefined in the same scope, it denotes the composite type of the type specified in its declaration and the previous type.

Additional Change 2b

6.7 Declarations

3 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that

– a typedef name may be redefined to denote the same type as it currently does **or to denote a function type which is the same type after adjustment of the arguments in both types according to 6.7.6.3, and** provided that type is not a variably modified type;

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