**N2862**
**Title: Wide Function Pointer Types for Pairing Code and Data (updates N2787)**
**Author:  Martin Uecker and Jens Gustedt**
**Prior Art: Borland C++, D**

**Changes since N2787:**

**- Corrections to some examples**
**- Limit proposal to C++ function qualifer syntax**
**- Add keyword**
**- Add abstract declarator syntax**
**- Mention new closures in libffi / GCC in introduction**
**- Add more examples related to existing extensions**

## Introduction:

Many existing C APIs require the use of void pointers that are passed as additional argument to callback functions and are therefor not type safe. Other languages often have native types which can be used to pass nested functions/lambda to callbacks. That there is no native C function type that pairs code and data makes interoperating with such languages difficult. Trying to use regular function pointers for callbacks with data requires run-time code generation (e.g. nested functions in GCC or XL C) or management of preallocated tables of trampolines (e.g. for closures in libffi [1] or recently proposed to GCC for Apple M1) but  is complex, inefficient, and – in case of run-time code generation - problematic from a security point of view. We therefor propose to add a wide function pointer type to C as an important missing basic building block for building type-safe APIs and for language interoperability.

## Prior Art

There exist many similar concepts in various languages. The version  is the __closure pointers in Borland C++ [2]

```
int (__closure *cb)(int x);
```

Another similar solution is also used in Apple’s blocks extension:

```
int (^cb)(int x);
```

A similar construct is std::function<> template in C++, but this is  a template type with additional features related to ownership and small-object optimization that do not seem to be implementable in C. A related feature for interfacing with other languages is GCC’s __builtin_call_with_static_chain() function. It  allows calling functions implemented in other languages that make use of a static chain from C code.

Many APIs in the industry already foresee the passing of additional information to function and lambda calls from other programming languages (e.g Pascal, Ada or Go) through a reserved hardware register. For example, for the System V AMD64 ABI this is the r10 register. From all 51 architectures for which support can be found in the GCC source tree all seem to support passing a static chain pointer. 48 use a register to pass the static chain pointer and three may use memory or choose a register or memory depending on the specific circumstances. The intent of this paper is to provide a standardized interface to such functionality.

## Proposal

We propose to add a wide function type similar to Borland C++ that pairs code and data (context). Pointers to such functions could then be used to call closures or callable objects from other languages. Such a pointer type could also be adopted by existing C extensions (blocks, nested functions) and be used in future C language extensions (e.g. lambdas) ensuring interoperability of APIs. A minimal proposal that would allow use of such pointers created in code from other languages could only add pointers to wide functions. To make this proposal useful also for programs written in C and to also capture some more functionality which now requires implementation-specfic assembler (as in libffi) or builtins (as in __builtin_call_with_static_chain), we also allow wide function definitions at file scope and add a simple API to manipulate the context of pointers to wide functions.

## Alternatives

Of course, users can implement their own structure type that combine code and data. But this has several downsides: 1. It can not be used with the standard ABI that uses a static chain register on most platforms. 2. It does not inter-operate with other languages while a new built-in type could be compatible with some existing ABIs. 3. Such a structure type would need to be defined for each function type separately. 4. Different libraries using such types would not be interoperabl.  5. Such type would not have implicit conversions that make it simple to integrate with existing code.

One could ask why implementations could not integrate the context pointer into existing function pointer types as the C standard does not require pointers to functions to have the same size as pointers to objects, or why the C pointer type could not point to a function descriptor that does include the context. In principle, this is possible and some platforms do exactly this. Unfortunately, for most platforms this would be an ABI breaking change. It also has run-time overhead which would then affect all function pointers.

## Implementation

How such pointers are implemented would be left entirely to the implementation. However, a sample implementation of a wide function pointer would use an internal representation as a pair of a code pointer and a context pointer:

```
struct wide_ptr_t {
  void (*code)();
  void *context;
};
```

We use here the type void(*)() as generic function pointer in the internal representation, but the type exposed in the C language for the pointed-to function has full type information about the return type and the arguments. The proposed library extension would provide functions to read and set the context pointer.

When the function is called the context pointer is passed in an ABI-specific way to the machine code pointed to by code. Typically, the context pointer is loaded into the static chain register which is already reserved by the ABI on most architectures as it is used by other languages such as Pascal or Ada (when using C this has to be done today using compiler extensions such as _builtin_call_with_static_chain). A regular function pointer can then be widened to a wide pointer simply by adding a NULL context pointer because loading this into the static chain register before calling a regular pointer would do no harm.

The feature is a powerful tool for interoperability as it can be adapted to call arbitrary callable entities in other languages from C regardless of the calling convention. The context pointer would point to the callable entity and the code pointer would point to some (compile-time) stub that knows how to call the entity with the right parameters.

## Proposed Semantics

It is suggested to implement the new approach by introducing a qualifier that applies to functions, i.e. introduce a new wide-qualified function type. The general rules for such wide functions are the  same as for regular functions. But introducing a new type for this allows us to introduce the extended functionality without changing the ABI of existing function pointer types. The use of a qualifier on functions is similar to how a qualifier on an object type would affect how an object is accessed during lvalue conversion and assignment. Similarly, a qualifier on a function would affect how a function is accessed, i.e. how it is called. We also need to specify the behavior for conversions: a regular function pointer can implicitly be converted to a wide function pointer (because this is a qualifier this works with existing language rules) and calling such a converted pointer should be allowed. Back conversion only yields a valid pointer if the wide function pointer was previously obtained by converting a regular function pointer, otherwise the behavior is undefined because the context has been lost. Two wide function pointers compare equal if and only if both code and context pointers compare equal in line with the idea that the context is part of the identity of a wide function.

**Proposed Syntax**

It is proposed to add a new keyword _Wide as a qualifier. Possible other names for the qualifier which could be considered are _Closure, _Callback, _Delegate.

Although this is currently useless, a qualifier can already be added to a function using a typedef:

```
typedef int foo_t(void);
const foo_const_t* ptr;
_Wide foo_wide_t *ptr;
```

Or added using typeof (if added to C23 or when available as an extension):

```
_Wide typeof(foo_t) *ptr;
```

This does not work for definitions. We propose to allow adding qualifiers to a function on the right (as C++ already allows for member functions).

```
int (*ptr)(void) _Wide;
```

Previously, we considered also the syntax for qualifiers using qualifier inside brackets (inspired by Borland C++ _Closure keyword), but based on feedback from WG14 we removed this version in this revision.

**Use with Existing Extensions**

A nested function in GCC or XL C could implicitly convert to a wide pointer with requiring the generation of a trampoline and would then be similar to the special function type used for Clang blocks – providing a portable type that could be used with these extensions and potentially with lambda expressions (if added to ISO C):

```
void foo(int (*f)(int) _Wide);

void outer_nested(int y)
{
  int bar(int x) { return x + y; }

  foo(bar);
}
```

```
void outer_blocks(int y)
{
  int (^bar)(int x) = ^(int x) { return x + y; };

  foo(bar);
}

struct S {
    int z;
    int bar(int x) { return x + z; }
};

void outer_member_fun(int y) // Borland C++
{
  S s = { y };

  foo(s.bar);
}

void outer_lambda(int y) // future C?
{
  int (*bar)(int x) _Wide = [y](int x) { return x + y; };

  foo(bar);
}
```

On some implementations, it is possible to set the static chain pointer for a call using a builtin function:

```
void foo(void (*f)(int x), void *env)
{
  __builtin_call_with_static_chain(f(0), env);
}
```

This could be written in a type-safe and portable way using the proposed interface:

```
void foo(void (*f)(int x) _Wide, void *env)
{
  f = wide_set_context(f, env);

  f(0);
}
```

**References**

[1] https://sourceware.org/libffi/
[2] http://docwiki.embarcadero.com/RADStudio/Sydney/en/Closure

**Proposed Wording**

**Change 1 (adding the _Wide qualifer)**

6.2.5 Types

(editorial note: insert after 29)
**30** A `_Wide` qualifier specifies a wide function type. A wide function is identified by a function definition and a context of type pointer to void. **XXX)**

(editorial note: footnote) XXX) Wide function types require implementation-specific calling conventions for setting up the context.

6.3.2.1

4 A function designator is an expression that has function type. Except when it is the operand of the sizeof operator, 71) or the unary & operator, a function designator with type "qualified or unqualified function returning type" is converted to an expression that has type "pointer to qualified or unqualified function returning type".

6.3.2.3 Pointers

8 A pointer to a function of one type may be converted to a pointer to a function of another type and back again; if the type pointed to by the intermediate pointer has all qualifiers of the original pointer, **t**he ~~result~~ back converted pointer shall compare equal to the original pointer, otherwise the behavior is undefined. ~~If a converted pointer is used to call a function whose type is not compatible with the referenced type, the behavior is undefined..~~

6.4.1 Keywords

Syntax

1 keyword: one of

… **_Wide** ...

6.5.1 Primary Expression

2 An identifier is a primary expression, provided it has been declared as designating an object (in which case it is an lvalue) or a function (in which case it is a function designator). 102) An identifier to a wide function that refers to a function definition

and that is used inside the execution of the wide function itself refers to the wide function which is currently called, i.e. the wide function with the same context as the context provided to the current call. Otherwise, an identifier for a wide function refers to the wide function which has a null pointer as context. YYY)

(editorial note: footnote) YYY) Thus, a recursive call to a wide function calls the same wide function, i.e. the wide function with the same function definition and context.

## 6.5.2.2 Function calls

9 If the function is defined with a type that is not compatible with the type (of the expression) pointed to by the expression that denotes the called function, the behavior is undefined, except that an expression with a wide-qualified type can be used to call a function of the corresponding unqualified type.

## 6.5.9 Equality operators

7 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space. 121) Two wide functions are considered the same function if and only if they have the same definition and the same context pointer.

## 6.7.3 Type qualifiers

1 Syntax

type-qualifier:
    const
    restrict
    volatile
    _Atomic
    **_Wide**

**Constraints**

(editorial note: insert after 4)
**5** The type modified by the _Wide qualifier shall be a function type.

## Semantics

**6** The properties associated with qualified types are meaningful only for expressions that are lvalues or that designate functions. 143)

**11** If the specification of an array type includes any type qualifiers, both the array and the element type is so-qualified. If the specification of a function type includes any type qualifiers other than the `_Wide` qualifier, the behavior is undefined. 149)

**16 Example 4** The following example illustrates how an old C API can be used underneath a new interface using a pointer to a wide function.

```
typedef void callback_old(void *data, int x);
extern void some_api_old(callback_old *cb, void *data);

typedef void callback_new(int x) _Wide;
extern void some_api_new(callback_new *cb);

static void stub(void *data, int x)
{
  (*(callback_new**)data)(x);
}

void some_api_new(callback_new *cb)
{
  some_api_old(stub, &cb);
}
```

**Change 2 (extended qualifier syntax – functions)**

6.7.6 Declarators

function-declarator:
        direct-declarator ( parameter-type-list$_{opt}$ ) **type-qualifier-list$_{opt}$**

6.7.6.3 Function declarators

Semantics

4 If, in the declaration " T D1 ", D1 has the form D ( parameter-type-list$_{opt}$ ) **type-qualifier-list$_{opt}$** attribute-specifier-sequence$_{opt}$ and the type specified for ident in the declaration " T D " is "derived-declarator-type-list T", then the type specified for ident is "derived-declarator-type-list function returning the unqualified version of T". **For each type qualifier in the list, ident is a so-qualified function.** The optional attribute specifier sequence appertains to the function type.
6.7.7 Type names

function-abstract-declarator:
        direct-abstract-declarator ( parameter-type-list$_{opt}$ ) **type-qualifier-list$_{opt}$**

**Change 3 a+b (lambdas)**

This change is conditional on the acceptance of lambdas in C23.

**6.3 Conversions**

**6.3.2.4 Lambda values**

A lambda value explicitly or implicitly converts to a pointer to wide function that has a compatible return type and a parameter list with the same order of compatible types. The pointer that is the result of such a conversion shall not be used with the wide_set_context generic function, see XXX.2. YYY) If the operand of the conversion is a function literal, the pointer is valid for the rest of the execution; if it is an lvalue of closure type, the pointer is valid until the end of the lifetime of the underlying closure object;

(for change 3a:)
otherwise the pointer is valid until the end of the full expression in which the conversion occurs.

(for change 3b:)

otherwise the pointer is valid until the end of the enclosing block, if any, or for the whole execution if the evaluation is performed in file scope.

YYY) Two pointers to wide function that are the result of different evaluations and conversion of the same closure expression may or may not compare equal, depending for example if the translator is able to prove that the values of captures are the same for any evaluation of the closure.

**Example** A lambda value can be converted to a pointer to  a wide function

```
int foo(int y, int z)
{
  int (*f)(int) _Wide = [y](int x) { return x + y; };
  return f(z + 3);
} // lifetime of lambda ends
```

### 6.5.3.2 Address and indirection operators

3 … If the operand of an address operator is a lambda value, the lambda value is converted to a wide function pointer of the appropriate type, see 6.3.2.4.

## Change 4 (library)

### XXX Wide functions `<stdwide.h>`

**1** The header `<stdwide.h>` declares two generic functions and the macro

```
  wide
```

that expands to _Wide.

2 In the following descriptions of the generic functions declared in the <stdwide.h> header T refers to an unqualified function type.

3 It is unspecified whether any generic function declared in <stdwide.h> is a macro or an identifier declared with external linkage. If a macro definition is suppressed in order to access an actual function, or a program defines an external identifier with the name of a generic function, the behavior is undefined.

## XXX.1 The `wide_get_context` generic function

### Synopsis

```
#include <stdwide.h>
void* wide_get_context(wide T *ptr);
```

### Description

The `wide_get_context` generic function returns the context of the wide function pointed to by ptr. If ptr is a null pointer, a null pointer is returned. If a pointer to an unqualified function is converted to a pointer to a wide function which is then passed to `wide_get_context`, a null pointer is returned.

### Returns

The `wide_get_context` generic function returns the context pointer or a null pointer.XXX)

(editorial note: footnote) XXX) If the `wide_get_context` generic function is called inside a wide function with its own identifier as argument, a pointer to its current context is obtained (cf. 6.7.3).

**Example** Inside a wide function,the `wide_get_context` generic function can be used to get access to its context.

```
struct bar { int y; };
int foo(int x) wide
{
  struct bar *context = wide_get_context(foo);
  return x + context->y;
}
```

## XXX.2 The `wide_set_context` generic function

### Synopsis

```
#include <stdwide.h>
```
wide T* wide_set_context(wide T *ptr, void *context);

### Description

The `wide_set_context` generic function returns a pointer to a wide function based on the wide function pointed to by ptr with the context changed to the value provided in the second argument.ZZZ) If a pointer to an unqualified function is converted to a pointer to a wide function and then passed to `wide_set_context`, the behavior is undefined. If ptr is a null pointer, a null pointer is returned.

ZZZ) The returned pointer is unique as long as the value passed to context is unique. If the wide_set_context generic function is called twice such that the arguments refer to the same wide function and the same object, the two return values compare equal.

**Returns**

The `wide_set_context` generic function returns a pointer to a wide function or a null pointer.

**Example 1** This example illustrates a C function interface which takes a pointer to void as context and that is implemented in terms of a new C API that makes use of a wide function type.

```
typedef void callback_old(void *data, int x);
extern void some_api_old(callback_old *cb, void *data);


typedef void callback_new(int x) wide;
extern void some_api_new(callback_new cb);


struct stub_data {
  callback_old *cb;
  void *data;
};

static void stub(int x) wide
{
  struct stub_data* context = wide_get_context(stub);
  context->cb(context->data, x);
}

extern void some_api_old(callback_old *cb, void *data)
{
  struct stub_data context = { cb, data };
  some_api_new(wide_set_context(stub, &context));
}
```

**Example 2**

```
typedef unsigned long counter_callback(void) wide;

static unsigned long counter_eval(void) wide
{
  unsigned long *count = wide_get_context(counter_eval);
  return ++(*count);
}

counter_callback* counter_create(void)
{
  unsigned long *count = calloc(1, sizeof *count);
  if (!count)
    return NULL;
  return wide_set_context(counter_eval, count);
}

void counter_delete(counter_callback *c)
{
  free(wide_get_context(c));
}

int main()
{
  counter_callback *my_counter = counter_create();
  if (!my_counter)
    abort();
  printf(“%ld\n”, my_counter());
  printf(“%ld\n”, my_counter());
  counter_delete(my_counter);
  return 0;
}
```