ISO/IEC JTC 1/SC 22/WG14 N 2067

August 20, 2016 v 1

The register overhaul
named constants for the C programming language

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

C11 lacks an important feature that would make everday's C programming and debugging much easier: named constants of any chosen data type. This proposal extends the existing **register** storage specification to provide that feature. As a fallout, some opitimization opportunities for addressless objects and functions also become available.

1. INTRODUCTION

The **register** storage class is perhaps the least understood, less esteemed and most underestimated tools of the C language. It merits better, since it can be used to force a very economic use of the & operator in code that is sensible to optimization. In particular, objects that are declared **register** can't alias, and, if they are **const** qualified in addition, they can often be completely optimized out.

The goals of this proposal are multiple:

GOAL 1. *Use* **const** *qualified* **register** *objects as typed compile time constants.*

GOAL 2. *Extend the optimization opportunities of* **register** *to file scope objects.*

GOAL 3. *Create new optimization opportunities for functions that are local to a TU.*

GOAL 4. *Improve the interplay between objects and functions that are local to a TU.*

GOAL 5. *Impose bounds checking for constant array subscripts.*

To see that the **register** concept can be extended to overcome the lack of general constants in C, let us look at the two following definitions:

```c
enum { fortytwo = 42, };
register int const fortytwo = 42;
```

From the point of view of the C standard, they specify very different things, an enumeration constant and an object. But as long as they are placed inside a function, the two entities are not much distinguishable by code that uses them.

— Both are of type **int**.
— Both have value 42.
— Both may not appear on the left hand side of an assignment.
— Both may not be subject of the address-of operator &.
— They have exactly the same spelling.

In C11, there are the following differences to these definitions:

— Definitions with **register** storage are not allowed in file scope.
— A **register** object with **const** qualification is not an *integer constant expression*, ICE, even if the initializer is an ICE.

Other drawbacks for **register** declarations in C11 include:

— Because of implicit address-of operations when accessing them, **register** is not useful for array declarations.

— Because of implicit address-of operations when calling functions, **register**, even if allowed in file scope, would not be useful for function declarations.

In view of that, I will try bring the concepts closer together, for which the separation is quite unproductive and artificial: literals and **const**-qualified **register** objects. Both are just constants in the sense of usual CS terminology.

> *A* constant *is an immutable and non-addressable entity in the program state that is determined during compilation.*

Because it is determined at compile time and immutable, no aspect of such a constant can change during a specific execution of the program. Because it is non-addressable no property of a constant can change between different executions of the same object file. Thus constants will never influence the branching of the program.

Therefore, constants can be present in a object file in very different forms.

— They may be optimized completely and only be implicitly present in the structure of the control flow graph.
— They can be present explicitly as one or several hardware registers, assembler immediates, or special instructions.
— They may be realized in addressable read-only memory.

Which form they take, depends on many factors and a compiler is completely free to chose. Examples:

— A `0` may be realized by an instruction that zeros out a register, *e.g* exclusive-or of the register with itself.
— Depending of its usage, a small integer constant can often be realized as instruction immediate, or fixed address offset.
— A **double** constant may be loaded directly from memory into a special floating point hardware register.

Overview

We develop our proposal in several steps, that become more and more intrusive as we go along. First in Section 2 we introduce the main feature, file scope **register**, which shows to be simple and straightforward. Section 3 then proposes to subsume the main advantage into a new concept, *named constants*, and Section 4 integrates these into C's *integer constant expressions*. Sections 5 and 6 then deal with two important features that are more difficult to integrate into C11, **register** functions and **register** arrays.

2. INTRODUCE REGISTER STORAGE CLASS IN FILE SCOPE

Unfortunately, with the current definition the use of **register** is limited to block scope and function prototype scope. There is no technical reason that it couldn't be used in file scope: file scope **register** declarations are already allowed from a viewpoint of syntax, they are explicitly forbidden by a constraint.

2.1. Changes for object types

Any compiler should be able to implement file scope *objects* with **register** storage class easily. At the minimum they can just be realized similar to file-scope-**static** with two additional features:

— Produce a diagnostic if the address of such an object is taken.
— If such a file scope **register** object is **const** qualified but not **volatile**, allow its use inside any **inline** or **register** function.

Textual changes for this feature that concern objects are minimal. The only problem spots are **const** qualified register objects. Since these are detectable at compile time, we can impose their explicit initialization. Thereby we avoid a clash with the concept of tentative definitions, which should just be forbidden for such **const** qualified objects.

Existing practice:

In connection with its `__asm__` hardware register extension, the GNU compiler already allows **register** in file scope.

Sections of the C standard to be amended

— 6.7.1 p6
  — Add a mention of aliasing to the optimization potential.
  — Add: *If such an object is of* **const** *qualified type, it shall be initialized explicitly.*
— 6.7.1 p6 footnote. Transform this footnote into a note.
  — Add after "**auto** declaration": "(function scope) or **static** declaration (file scope)"
  — Add: *A* **const** *qualified identifier with* **register** *storage class can not appear in a tentative definition.*
— 6.9 p2 remove the mention of **register**
— 6.9.1 p4 add **register** to the list
— 6.9.2 p2 add **register** to the cases

3. TYPED CONSTANTS WITH REGISTER STORAGE CLASS AND CONST QUALIFICATION

The idea of this proposal is that named constants for all types become available in file scope, because they can then be declared in header files as **const** qualified **register** objects.

No additional changes to the ones in the previous section are *required*. It might be convenient, though, to add some text to stress the fact that also a hidden modification of such objects is not standard conforming. This may be helpful to avoid that implementations claim the right to map and modify such objects, hiding behind the fact that the standard leaves behavior undefined when a **const** qualified object is modified.

Sections of the C standard to be amended

— **6.6** "constant expressions", **p7** general constant expressions, add an item to the end of the list
— an lvalue of **const** but not **volatile** qualified type that has been declared with **register** storage class.

— Baptize the thing. Therefore add a new paragraph:
A *register constant* is an object that is of **const** but not **volatile** qualified type, that is declared with the **register** storage class, for which the unique declaration is the definition, that is explicitly initialized and for which the initializer only contains constant expressions.(FOOTNOTE1) Such a register constant provides the same value as specified by the initialization throughout all its lifetime.(FOOTNOTE2)
FOOTNOTE1: Register constants can not appear in tentative definitions.
FOOTNOTE2: But for their spelling, for their type qualification, [*for their qualification as an ICE,*] and for their usability in preprocessor expressions, register constants are indistinguishable from literals of the same type and value. Therefore, register constants can be be used as named constants of their type.

— **p8**, arithmetic constant expressions: Add *register constants of arithmetic type* to the list.

— **6.7.1** "storage class specifiers" **p6**: new footnote:
An implementation should only map a **register** declared object to a hardware register or similar device which is subject to changes not effected by the program if the type of the object is **volatile** qualified and therefore is not a register constant.

— **6.7.4** "function specifiers", **p3**. Allow the use of register constants in all **inline** functions. Therefore at the end add:
... shall not contain a reference to an identifier with internal linkage unless it is the name of a register constant.

4. EXTEND ICE TO REGISTER CONSTANTS

Now that we have global constants for all types, we need to integrate this into the rest of the language. We want register constants to behave the same as literals of the underlying type. This is particularly important for constants of integer type, since they are needed to declare array dimensions, alignments, width of bit-fields or values of enumeration constants, and we want all of this to go smoothly.

To make this possible, we have to amend the computation of compile time integer constant, or as the C standard calls them *integer constant expressions*, ICE. In most contexts, attaching the ICE property to a C11 expression will not change the semantics if this new **register** feature is applied.

There are only two such contexts in which **register** objects were previously allowed, and where the meaning changes if the ICE property is added:

— An array declaration declares a VLA if the expression that is used for the size is not an ICE. Thus some VLA that are declared in C11 code would turn into FLA. Such a change would not change the semantics of the program in question.
— Integer expressions of value `0` are only null pointer constants if they are also ICE.

There is no problem by using pointer expressions directly.

```
register int const zero = 0;
double* p = zero;             // constraint violation in C11
```

In C11, the initialization of `p` is a constraint violation because implicit conversions from integer to pointer expressions are only allowed if the integer expression is an ICE of value `0`. So this property for pointers alone would just have code valid that had been invalid C11 code, before.

Only a combination of some rarely used features may effectively result in a change of semantics of a valid C11 program. It requires a very specific use of **const** qualified **register** objects that are used in a very specific **_Generic** primary expression. These situations are detectable at compile time and during the transition phase from C11 to C2x compilers can implement diagnostics for this situation.

Sections of the C standard to be amended
The change itself is quite easy to perform, because we already have identifiers in C11, namely enumeration constants, that are ICE.

— In all places that it occurs in C11, change the syntax term *enumeration-constant* to *named-constant*. These are **6.4.4, p1**, **6.4.4.3**, **6.7.2.2 p1**, **A.1.5**.
— **6.4.4.3**, previously “enumeration constants”: Change the whole section to read

```
    6.4.4.3 Named constants

    Syntax
                 named-constant: identifier

    Semantics

    Named constants are identifiers that are declared as register
    constant or as enumeration constant. An identifier declared
    as an enumeration constant has type int.

    Forward references: constant expressions (6.6), enumeration
    specifiers (6.7.2.2).
```

— **6.6** “constant expressions”, **p6** “ICE” and footnote: change *enumeration constant* to *named constant of integer type.*
— **p8** “arithmetic constant expression”: change *enumeration constant* to *named constants of arithmetic type.*

5. FUNCTIONS

The possibility to declare functions with **register** storage class is an interesting fallout of our approach.

— A function declaration of storage class **register** is equivalent to a declaration as **static inline** with the additional property that its address can’t be taken.

This doesn't mean that implementors of that feature have to implement a completely new function model. Just as currently for **register** variables, a **register** function *may* well reside in memory and internally the compiler can use its address to make a call. But such a mechanism would be to the discretion of the implementation and completely transparent for the programmer.

5.1. Optimization opportunities

The advantages of **register** functions are:

— Such a function cannot be called from another TU than the one it is defined in and can effectively be inlined without negative effects to other TU. Thus no "*instantiation*" of the symbol of a **register** function is necessary.
— Since the caller of such a function is known to reside in the same TU, the function setup can avoid the reload of certain registers and is not bound to the platform's function call ABI.

Similar as for **register** objects, these optimizations are currently possible for **static inline** functions for which the compiler can prove that the address never escapes the current TU. A **register** declaration instead of **static inline** guarantees that these optimization opportunities are not lost accidentally.

Existing practice:

The GNU compiler already has

```
__attribute__((__visibility__("internal")))
```

that allows for similar optimizations, but which is intrinsically unsafe since gcc does not check if a pointer to such a function can escape the current TU.

5.2. Relaxed constraints for TU local objects

As already mentioned, **inline** declared functions have difficulties with other symbols that have no or internal linkage. Such functions that are not **static** at the same time

— cannot access **static** file scope variables, or
— cannot declare their own block scope **static** variables,

even if these are **const** qualified and known at compile time.

Our proposal simplifies things with that respect.

— All functions that are **static inline** have access to **register** objects that are visible at their point of definition.
— All functions declared **inline** have access to register constants that are visible at their point of definition.

5.3. Changes for function types

As the function concept is currently formulated, such **register** function could never be called: the standard function call feature takes the address of a function that is called. We propose to change the standard to allow for that by letting the function call operator `()` directly operate on a function. This changes the semantics just in the context of a call. Function to function-pointer would still be mandated in all other contexts and thus would be illegal for functions that are declared with **register**.

Sections of the C standard to be amended

- **6.3.2.1 p4** "conversions of function designators": Add ", *or as the postfix expression of a function call*" to the list of allowed operations.
- **6.5.2.2** "function calls":
  - **p 1**: Replace

    ... pointer to function returning void or returning a complete object type other than an array type.

    by

    ... function returning void or returning a complete object type other than an array type, or shall have type pointer to such a function.
  - **p5**: Replace

    ... type pointer to function returning an object type, ...

    by

    ... type function returning an object type or pointer to such a function, ...

6. UNIFY DESIGNATORS

If we want to extend the use of register constants, we want to ensure that the types that can be used for it are not artificially constrained.

With

```
enum sig_stat { unknown = 0, sync = 1, async = 2, };
```

consider the two following alternative declarations of `sig_status` in a header file:

```
extern enum sig_stat const sig_status[];    // values are hidden
```

and

```
static enum sig_stat const sig_status[] = { // not usable in inline
  [SIGABRT] = async,
  [SIGFPE]  = sync,
  [SIGILL]  = sync,
  [SIGINT]  = async,
  [SIGSEGV] = sync,
  [SIGTERM] = async,
};
```

The first hides the contents from the user code, and the fact if `sig_status[SIGIMPL]` is `unknown` or not[1] can only be checked at runtime, through a memory reference. The second declaration exposes all values to all users, but cannot be used from an **inline** function.

Using the second variant with **register** instead of **static** would solve the problem: all values are visible for all users and all **inline** functions could use it. But unfortunately, with the current version of the standard, arrays that are declared with a **register** storage class can't serve much purpose: we can't even access individual fields without constraint violation. This is because the `[]` designator is defined by taking addresses: refering to an array element through `A[23]` is equivalent an address-of operation, pointer arithmetic and

[1]for some implementation defined signal `SIGIMPL`

dereferencing, namely `*(&A[0]+23)`. Thus if `A` is declared as register, this is a constraint violation.

This part of the **register** overhaul tries to make **register** arrays useful whenever the expression in the `[]` is an ICE. It does that by attempting a careful review of the three different syntactical contexts, in which `[]` brackets are used.

Syntactically, C uses `[]` brackets in three different places, but in all that places they serve to specify a length (or the lack of) an array dimension or position:

— array declaration,
— array initialization,
— array subscripting.

The expression that is enclosed inside `[]` can be

— a strictly positive integer constant expression (ICE), or empty `[]` to provide a constant value that is determined at compile time
— any other form of integer expression, or empty `[*]` to provide a value that is determined at run time.

All actual C compilers are able to distinguish these two cases. For designated initializers, only the first form is allowed and using the second is a constraint violation. For an array declaration, in the first case the array is FLA, in the second a VLA[2].

For subscripting, the C standard currently doesn't make the distinction. This is a bit unfortunate because conceptually subscripting an array at fixed location is technically the same as accessing an element of a structure. In this section we aim to amend the C standard such that an subscript expression that is a positive ICE can be applied to a **register** array.

As a fallout, this reorganization ensures bounds checking for FLA and constant subscripts. If an index into an FLA is constant it can be checked against `0` and the array length at translation time. So if it is negative or too large this constitutes a constraint violation under this proposal.

Existing practice:

The `clang` compiler silently implement this already, although this is not compliant to C11.

---

[2]Which may or many not be supported by the compiler.

Sections of the C standard to be amended

```
   6.5.2 Postfix operators

   Syntax

   postfix-expression:
      primary-expression
      postfix-expression array-designator
      postfix-expression [ expression ]
      postfix-expression ( argument-expression-listopt )
      postfix-expression struct-union-designator
      postfix-expression -> identifier
      postfix-expression ++
      postfix-expression --
      ( type-name ) { initializer-list }
      ( type-name ) { initializer-list , }

   array-designator: [ expression ]

   struct-union-designator: . identifier

   Constraints
```

   The *expression* of an *array-designator* shall be an integer
   constant expression.

```
 6.5.2.1 Array subscripting

 Constraints
```

Array subscripting can occur in two different forms. For the first, the
first expressions has type *array object of type* T *and constant length* L,
for some type T non-negative integer constant L and is
not a variable length array. The integer constant expression of the
array-designator shall have a non-negative value that is strictly
less than L. The result has type T.

Otherwise, after promotion [the first | one] expression shall have type
pointer to complete object type T, the other expression
shall have integer type, and the result has type T.

```
Semantics

A postfix expression followed by an array-designator designates
the corresponding element of an array object.FOOTNOTE Successive
array-designators designate an element of a multidimensional array
object.

FOOTNOTE: Because the subscript expression is a integer constant such a
designation is similar to the designation of structure or union members.

Otherwise, --- insert the existing text about array subscript semantics ---

 EXAMPLE Consider the array object defined by the declaration

  int x[3][5];

  Here x is a 3 × 5 array of ints; more precisely, x is an
  array of three element objects, each of which is an array of
  five ints. So the expression x[1] denotes an array object of
  five int, namely the second such array element. When used in
  the expression x[1][2], the designator [2] applies to that
  array and thus refers to its third element, an int.

--- insert existing array subscript example here ---

Forward references: Structure and union members (6.5.2.3),
additive operators (6.5.6), address and indirection operators (6.5.3.2),
array declarators (6.7.6.2), designator (6.7.9).
```

In Section 6.7.9 and in Annex A, change the syntax definition for *designator* to the following:

```
designator: array-designator struct-union-designator
```

Also change p6 and p7 of that section to directly refer to *array-designator* and *struct-union-designator*, respectively.