The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.1 Storage-class specifiers

storage-class specifier syntax

1364

```
storage-class-specifier:
                typedef
                extern
                static
                auto
                register
```

Commentary

abbreviating identifier

The character sequence *regist* is not generated by the algorithms commonly used by speakers of English for abbreviating *register*.

C++

The C++ Standard classifies **typedef** (7.1p1) as a *decl-specifier*, not a *storage-class-specifier* (which also includes **mutable**, a C++ specific keyword).

Other Languages

Languages often use the keyword **type** to denote that a type is being declared, although a few languages (e.g., Algol 68 and CHILL) use **mode**, or some variation of that word. Java is unusual, in a modern language, in not providing a mechanism for defining a name to have a primitive (scalar in C terminology) type or array type.

Some languages also use the keyword **extern**. Fortran uses the keyword **extern** to declare a parameter as denoting a callable function (it does not have function types as such).

Some languages (e.g., CHILL) provide a mechanism for specifying which registers are to be used to hold objects (CHILL limits this to the arguments and return value of functions). The keyword **register** is unique to C (and C++).

Pascal requires that the keyword **forward** on procedure and function declarations that are defined later in the same source file.

Coding Guidelines

footnote 100 1372
for statement declaration part
external declaration not auto/register
automatic storage duration

The standard only uses the keyword **auto** in a few places (outside of comments in examples). However, the phrase *automatic storage duration* occurs much more often. An occurrence of the keyword **auto** in the visible source provide little additional information to a reader. A declaration's lexical position with respect to being inside/outside of a function definition, or the presence of other storage-class specifiers provides all the information required by a reader. As the Usage figures show (see Table 1364.1) existing practice is not to use this keyword. A guideline recommending against its use would be redundant.

Table 1364.1: Common token pairs involving a *storage-class*. Based on the visible form of the `.c` files (the keyword **auto** occurred 14 times).

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
static void	33.7	32.7	**extern int**	32.1	1.7
static int	28.2	15.1	**register struct**	19.1	1.4
typedef union	3.2	11.0	**typedef struct**	62.4	1.2
static const	1.5	10.0	**register int**	23.0	1.2
static volatile	0.3	8.6	**register char**	10.2	1.2
typedef enum	10.8	8.2	**register unsigned**	6.1	0.9
static signed	0.0	6.5	**extern char**	7.4	0.9
static unsigned	3.8	5.5	**extern struct**	6.9	0.5
extern double	1.3	5.5	**static** identifier	21.0	0.3
static char	4.1	5.1	**typedef unsigned**	6.2	0.2
static struct	6.4	4.8	**typedef** identifier	7.9	0.0
register enum	1.6	4.6	**register** identifier	35.9	0.0
extern void	21.5	2.1	**extern** identifier	23.7	0.0

Table 1364.2: Common token pairs involving a *storage-class*. Based on the visible form of the `.h` files (the keyword **auto** occurred 6 times).

Token Sequence	% Occurrence First Token	% Occurrence of Second Token	Token Sequence	% Occurrence First Token	% Occurrence of Second Token
typedef union	12.4	67.1	**typedef unsigned**	6.6	3.1
typedef enum	6.2	37.2	**extern unsigned**	2.9	2.8
typedef signed	0.5	28.6	**static void**	10.3	2.2
extern void	28.6	24.0	**typedef void**	4.0	1.6
extern double	0.3	17.9	**static int**	7.0	1.2
typedef struct	46.3	16.6	**extern** identifier	32.2	0.9
extern int	23.2	15.2	**register long**	16.0	0.8
extern float	0.3	9.8	**register unsigned**	24.8	0.6
register signed	2.6	8.2	**static** identifier	70.3	0.5
static const	6.4	5.0	**register int**	18.4	0.3
extern char	3.8	4.8	**typedef** identifier	16.7	0.2
extern struct	4.3	3.3	**register** identifier	18.4	0.0

Constraints

1365 At most, one storage-class specifier may be given in the declaration specifiers in a declaration.[100)]

Commentary

storage duration object linkage

The storage-class specifier can be used in a declaration to specify two attributes: the storage duration, and linkage. There is only one context where the appearance of a storage-class specifier in a declaration can affect the storage duration of the object being declared. An object declared in block scope has automatic storage duration unless either of the keywords **extern** or **static** appear in its declaration. In which case it has static storage duration. The presence of the storage-class specifiers **typedef**, **extern**, and **static** may cause the default linkage given to an identifier, because of where it is declared in the source, to be changed.

C++

While the C++ Standard (7.1.1p1) contains the same requirement, it does not include **typedef** in the list of *storage-class-specifiers*. There is no wording in the C++ limiting the number of instances of the **typedef** *decl-specifier* in a declaration.

Source developed using a C++ translator may contain more than one occurrence of the **typedef** *decl-specifier* in a declaration.

Other Languages

In many languages the location of an objects declaration in the source code is used to specify its storage class. Like C, some languages provide keywords that enable the default storage class to be overridden, e.g., Fortran **common**.

Coding Guidelines

Should a storage-class specifier ever appear in a declaration? The only two that are ever necessary are **typedef** and **static** (to change default behavior; **extern** in block scope is not necessary because the declaration can be moved to file scope). All other uses are related to coding guidelines issues. The use of **extern** is covered by the guideline recommendation dealing with a single point of declaration.

identifier declared in one file ??

The storage duration and linkage issues associated with the storage-class specifier **static** are discussed elsewhere. The efficiency issues associated with the storage-class specifier **register** are discussed elsewhere.

static internal linkage
static storage duration
register storage-class 1369

Semantics

The **typedef** specifier is called a "storage-class specifier" for syntactic convenience only; 1366

Commentary

The keyword **typedef** is not really a storage class as such. However, the syntax for typedef name declarations is the same as that for object and function declarations. The committee considered it a worthwhile simplification to treat **typedef** as a storage class.

C++

It is called a *decl-specifier* in the C++ Standard (7.1p1).

Other Languages

In other languages the keyword used to define a new type is rarely placed in the same syntactic category as the storage-class specifiers.

it is discussed in 6.7.7. 1367

The meanings of the various linkages and storage durations were discussed in 6.2.2 and 6.2.4. 1368

register storage-class

A declaration of an identifier for an object with storage-class specifier **register** suggests that access to the object be as fast as possible. 1369

Commentary

It is the developer who is making the *suggest*ion to the translator (sometimes the term *hint*) is used). The original intent of this storage-class specifier was to reduce the amount of work a translator vendor needed to do when implementing a translators machine code generator (which also contributed to keeping the overall complexity of a C translator down). This suggestion is often interpreted by developers to mean that translators will attempt to keep the values of objects declared using it in registers. Had C been designed in the 1990s the keyword chosen might have been **cache**.

cache
external declaration not auto/register

The standard does not permit declarations at file scope to include the storage-class specifier **register**.

C++

7.1.1p3 *A* ***register*** *specifier has the same semantics as an* ***auto*** *specifier together with a hint to the implementation that the object so declared will be heavily used.*

Translator implementors are likely to assume that the reason a developer provides this hint is that they are expecting the translator to make use of it to improve the performance of the generated machine code. The

C++ hint does not specify implementation details. The differing interpretations given, by the two standards, for hints provides to translators is not likely to be significant. The majority of modern translators ignore the hint and do what they think is best.

Other Languages

While it might be possible to make use of particular techniques (e.g., defining the most frequently accessed objects first) to improve the quality of machine code generated by translators of any language, the specification of a language's semantics rarely includes such hint mechanisms.

Common Implementations

Processors rarely perform operations directly on values held in storage, they move them to temporary working locations first. In nearly all cases these temporary working locations are a small set of storage areas, known as *registers*, in the processor itself. Some processors have an architecture that is stack-based[10] rather than register-based and the Bell Labs C Machine[6] was a stack-based processor specifically designed to execute C. However, experience with the two architectural choices has shown that the register-based approach yields better overall performance and the majority of modern processes use it. Stack-based processors are not as common as they once were, although new ones are still occasionally built.

expression processor evaluation

What is a register? The documentation for the Ubicom IP2022[15] says it has 255 registers. However, the transistors used to represent the values of these registers (at least in the current implementation) are in the same area of the chip as the rest of storage. In fact these *registers* occupy the lower portion of the processors address space. They are accessed using a direct addressing mode (the same could be said about registers in other processors, except that rarely share an address with the rest of storage).

Customer demand for higher-performance (in the generated machine code) and competition between translator vendors means that vendors desire to minimize the cost of producing a code generator no longer includes not implementing a sophisticated register allocator.

Using the storage-class specifier **register** to help improve the performance of the generated machine code can be a difficult process that sometimes has the reverse affect (programs that execute more slowly). Translator implementors are aware that developers expect objects declared using **register** to have their values kept in registers when possible. Some implementors have decided to attempt to meet this expectation, for the duration of the objects lifetime. The result can be a decrease in performance, because a register is used to hold a particular objects value during a sequence of statements where it would have been better for that register to be holding a different objects value. Other translator implementors believe that the known register allocation algorithms produce better results and ignore any developer provided hints.

The values of objects are not the only things that may be worth trying to keep in registers. If a sequence of code performs the same operation on two operands, whose values have not changed between the two occurrences (a common subexpression), keeping the result in a register and reusing it may be more efficient than performing the operation again. Use of the **register** storage-class specifier can have an indirect benefit. An optimizer does not need flow analysis to deduce that the object is not aliased; it is not permitted to as the operand of the address-of operator.

common subexpression

unary & operand constraints

register optimizing allocation

A number of algorithms have been proposed for allocating values to registers (it is known that optimal register allocation is NP-hard[7]). An equivalence between this problem and the pure mathematics problem of graph coloring has been shown to exist.[5] So called *register coloring* algorithms have proved to be very popular for one class of processors (those having many registers that are treated orthogonally) and a variety of different variations on this approach have been used. When the time taken to translate code is important (e.g., just in time compilation) *linear scan* register allocation[12] is much faster than register coloring and has been found to result in code that executes only 12% slower than an aggressive register allocator.

expression optimal evaluation

It has proved difficult to find general algorithms for register allocation on processors having few registers, or where there are restrictions on the operations that can be performed using some registers. In many cases the allocation algorithms are hand tailored for each processor. Mapping the problem to one in integer linear programming has produced worthwhile results for processors having few registers, such as the Intel Pentium.[2]

array element held in register

Register allocation algorithms generally only consider trying to maintain objects having a scalar type in registers. A study by Li, Gu, and Lee[11] evaluated the benefits of doing the same for array elements, known as *scalar replacement* (see Chapter 8 of Allen[1]). They used trace driven simulations of a variety of Fortran benchmark programs to obtain idealised (tracing the actual data references) measurements to obtain the best results that could be achieved, if an optimizer optimally assigned objects to registers. The results showed that having between 48 and 96 registers available, for holding scalar and array element values, had the most impact (savings were even possible with 32 available registers). On five C based multimedia kernels an implementation of scalar replacement by So and Hall[14] reduced memory accessed by 58% to 90% and saw speedups of between 2.34 and 7.31.

A number of studies have investigated the affects of increasing or decreasing the number of registers available to a translator on the performance of the generated machine code. Having sufficient registers to be able to keep all object values in one of them may be the ideal. In practice optimizers are limited by the analysis they perform on the source. For instance, if it is not possible to deduce whether a particular object is referenced via a pointer, an optimizer has to play safe and access the value from storage. Performance improvements have been found to more or less peak at 17 registers using lcc,[4] later analysis using more sophisticated optimizations (including inlining and allocating globals to registers) found that effective use could be made of 64 registers.[13]

Many early processors tended to have few registers because of hardware complexity and cost considerations. The relative importance of these considerations has decreased over time and having 32 registers is a common theme among modern processors. However, in practice once function calling conventions are taken into account and various registers reserved (for a variety of reasons, for instance holding the stack pointer) there are rarely more than 16, out of 32, truly temporary registers available to a translator.

ABI
external declaration
not auto/register

Some implementations support the use of the **`register`** storage-class specifier on declarations at file scope. Such declarations are usually specifications of which register should be dedicated to holding a particular object.[3,9]

Franklin and Sohi[8] measured register usage while programs from the SPEC89 benchmark executed on a MIPS R2000 (see Table 1369.1).

Table 1369.1: Degree of use of floating-point and integer register instances (a particular value loaded into a register). Values denote the percentage of register instances with a particular degree of use (listed across the top), for the program listed on the left. For instance, 15.51% of the integer values loaded into a register, in gcc, are used twice. Left half of table refers to floating-point register instances, right half of table to integer register instances. Zero uses of a value loaded into a register occur in situations such as an argument passed to a function that is never accessed. Adapted from Franklin and Sohi.[8]

Usage	0	1	2	3	≥4	Average	0	1	2	3	≥4	Average
eqntott							0.89	71.34	17.54	9.47	0.76	1.86
espresso							3.67	72.30	17.66	3.74	2.63	1.48
gcc							6.26	67.37	15.51	4.45	6.41	1.69
xlisp							4.27	66.14	12.42	10.20	6.97	1.84
dnasa7	0.00	99.83	0.02	0.03	0.12	1.31	0.67	2.36	16.29	64.36	16.33	3.28
doduc	1.46	84.00	9.51	1.94	3.09	1.36	10.31	44.35	26.52	10.13	8.69	2.93
fpppp	0.16	91.09	6.15	1.14	1.46	1.16	1.34	10.12	83.45	0.46	4.63	3.09
matrix300	0.00	99.92	0.00	0.00	0.08	1.25	15.29	61.54	7.71	0.12	15.35	1.92
spice2g6	0.21	79.85	19.22	0.16	0.56	1.22	4.04	73.38	12.08	3.56	6.94	1.68
tomcatv	0.00	86.43	8.30	1.49	3.77	1.26	0.12	24.99	37.54	27.40	9.96	3.22

register extent effective

The extent to which such suggestions are effective is implementation-defined.101) 1370

Commentary

All translators need to have a register allocation algorithm to be able to generate executable machine code. The extent to which the **`register`** storage-class specifier affects the behavior of this algorithm needs to be documented. A translator's documented behavior is unlikely to be sufficient to enable a developer to predict which values will be held in which processor register. The possible permutations are rarely sufficiently small that the behavior is easily enumerated.

C++

The C++ Standard gives no status to a translator's implementation of this hint (suggestion). A C++ translator is not required to document its handling of the **register** storage-class specifier and often a developer is no less wiser than if it is documented.

Coding Guidelines

Those coding guideline documents that base their recommendations on the list given in annex I are likely to recommend against the use of the **register** storage-class specifier. The only externally visible affect of using the **register** storage-class specifier is a possible change in execution time performance or size of program image. In both cases the changes are unlikely to be worth a guideline.

program image

Example

Macros provide a flexible method of controlling the definitions that contain the **register** storage-class specifier.

```
#if EIGHT_BIT_CPU != 0
#define REG1 register
#define REG2
#define REG3
#define REG4
#endif

#if MODERN_DSP != 0
#define REG1 register
#define REG2 register
#define REG3
#define REG4
#endif

#if RISC_CHIP != 0
#define REG1 register
#define REG2 register
#define REG3 register
#define REG4 register
#endif

void f(void)
{
REG1 int total_valu;
REG2 short intermediate_valu;
REG3 int the_valu;
REG4 long less_often_used_value;

/* ... */
}
```

1371 The declaration of an identifier for a function that has block scope shall have no explicit storage-class specifier other than **extern**.

block scope storage-class use

Commentary

There is a lot of existing source containing function declarations at block scope using the **extern** storage-class specifier. The Committee did not want to render such usage as undefined behavior and decided to permit it.

Other Languages

Most languages do not support the declarations of identifiers for functions in block scope. Although some languages do support nested function definitions.

Common Implementations

Translators usually flag an occurrence of this undefined behavior.

Coding Guidelines

identifier declared in one file ??

If the guideline recommendation dealing with function declarations at file scope is followed this requirement is not an issue.

---

footnote 100

100) See “future language directions” (6.11.5). 1372

---

footnote 101

101) The implementation may treat any **register** declaration simply as an **auto** declaration. 1373

Commentary

unary & operand constraints

That is to say, an implementation may treat such a declaration as an **auto** declaration for the purposes of storage allocation. However, the various constraints and other kinds of behavior associated with an object declared using the **register** storage class still apply.

However, whether or not addressable storage is actually used, the address of any part of an object declared with storage-class specifier **register** cannot be computed, either explicitly (by use of the unary **&** operator as discussed in 6.5.3.2) or implicitly (by converting an array name to a pointer as discussed in 6.3.2.1). 1374

Commentary

unary & operand constraints

A constraint violation occurs if the operand of the address-of operator has been declared using storage-class specifier **register**.

type category

This observation applies when the type category is an array type. But, it does not apply to the case where a member of a structure or union type has an array type. For instance:

```
struct {
       register int mem[100];
       } x;
&x.mem;    /* &     applied to an array type. */
&x;        /* & not applied to an array type. */
```

C++

unary & operand constraints

This requirement does not apply in C++.

Thus, the only operator that can be applied to an array declared with storage-class specifier **register** is **sizeof**. 1375

Commentary

array converted to pointer

unary & operand constraints

An operand having an array type is not converted to pointer type when it is operated on by the address-of or **sizeof** operator. The former would be a constraint violation, leaving the latter.

C++

unary & operand constraints

This observation is not true in C++.

If an aggregate or union object is declared with a storage-class specifier other than **typedef**, the properties resulting from the storage-class specifier, except with respect to linkage, also apply to the members of the object, and so on recursively for any aggregate or union member objects. 1376

Commentary

Members of a structure or union type always have no linkage. member no linkage

Another property resulting from the presence of a storage-class specifier is the storage duration of an object. The other properties might more properly be called *consequences*. The consequences arising from wording in other parts of the Standard, such as Constraints, undefined and implementation-defined behaviors. For instance, while placing the **register** storage-class specifier on the declaration of an object having a structure type may not result in any of its members being held in processor registers, the associated constraints still apply (it is a constraint violation for the result of a member selection operator to appear as the operand of the address-of operator). storage duration object 1369 register storage-class

C90

This wording did not appear in the C90 Standard and was added by the response to DR #017q6.

C++

The C++ Standard does not explicitly specify the behavior in this case.

Other Languages

In many object-oriented languages it is possible to specify that members of classes have a different storage-class, or linkage, than the class itself.

1377 **Forward references:** type definitions (6.7.7).

References

1. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architecture*. Morgan Kaufmann Publishers, 2002.
2. A. W. Appel and L. George. Optimal spilling for CISC machines with few registers. Technical Report TR-630-00, Princeton University, Mar. 2001.
3. ARM. *ARM Developer Suite: Compilers and Libraries Guide*. ARM Limited, 1.2 edition, Nov. 2001.
4. M. E. Benitez and J. W. Davidson. Register deprivation measurements. Technical Report CS-93-63, Department of Computer Science, University of Virginia, Nov. 15 1993.
5. P. Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Apr. 1992.
6. D. R. Ditzel and H. R. McLellan. Register allocation for free: The C machine stack cache. In *Proceedings of the First International Symposium on Architectural support for Programming Languages and Operating Systems*, pages 48–56, 1982.
7. M. Farach and V. Liberatore. On local register allocation. Technical Report DIMACS Technical Report 97-33, Rutgers University, July 1997.
8. M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25^{th} Annual International Symposium on Microarchitecture (MICRO-25)*, pages 236–245, 1992.
9. T. Instruments. *TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide*. Texas Instruments, Inc, spru024e edition, Aug. 1999.
10. P. J. Koopman Jr. *Stack Computers the new wave*. Mountain View Press, 1989.
11. Z. Li, J. Gu, and G. Lee. An evaluation of the potential benefits of register allocation for array references. In *Proceedings of the 1^{st} Workshop on Interaction between Compilers and Computer Architectures*, Feb. 1996.
12. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, Sept. 1999.
13. M. Postiff, D. Greene, and T. Mudge. The need for large register files in integer codes. Technical Report CSE-TR-434-00, The University of Michigan, Aug. 2000.
14. B. So and M. W. Hall. Increasing the applicability of scalar replacement. In *Compiler Construction, 13th International Conference, CC 2004*, pages 185–201, Mar. 2004.
15. Ubicom. *IP2022 Internet Processor*. Ubicom, Inc, preliminary edition, July 2002.