The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.2.4 Storage durations of objects

storage duration object

An object has a *storage duration* that determines its lifetime. 448

Commentary

Storage duration is a property unique to objects. In many cases it mirrors an object's scope (not its visibility) and developers sometimes use the term *scope* when *lifetime* would have been the correct term to use.

C++

1.8p1 *An object has a storage duration (3.7) which influences its lifetime (3.8).*

In C++ the initialization and destruction of many objects is handled automatically and in an undefined order (exceptions can alter the lifetime of an object, compared to how it might appear in the visible source code). For these reasons an object's storage duration does not fully determine its lifetime, it only influences it.

Other Languages

Most languages include the concept of the lifetime for an object.

There are three storage durations: static, automatic, and allocated. 449

Commentary

One of the uses of the overworked keyword **`static`** is to denote objects that have static storage duration (there are other ways of denoting this storage duration).

C90

The term *allocated* storage duration did not appear in the C90 Standard. It was added by the response to DR #138.

C++

3.7p1 *The storage duration is determined by the construct used to create the object and is one of the following:*

— *static storage duration*

— *automatic storage duration*

— *dynamic storage duration*

The C++ term *dynamic storage* is commonly used to describe the term *allocated storage*, which was introduced in C99.

Common Implementations

stack

Objects having particular storage durations are usually held in different areas of the host address space. The amount of static storage duration is known at program startup, while the amounts of automatic and allocated storage varies during program execution. The most commonly seen division of available storage is to have the two variable-size storage durations growing toward each other. Objects having static storage duration are often located at the lowest address rather than the highest. This design decision may make it possible to access these objects with a single instruction using a register + offset addressing mode (provided one is available).

register + offset

A few implementations do not have separate stack and heap areas. They allocate stack space on the heap, on an as-needed basis. This usage is particularly common in realtime, multiprocess environments, without hardware memory management support to map logical addresses to different physical addresses. The term *cactus stacks* is sometimes used.

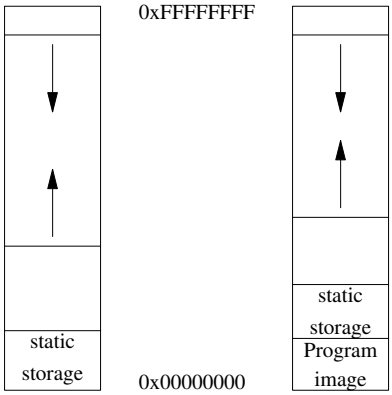

Figure 449.1: The location of the stack invariably depends on the effect of a processor's pop/push instructions (if they exist). The heap usually goes at the opposite end of available storage. The program image may, or may not, exist in the same address space.

Some processors (usually those targeted at embedded systems[10]) support a variety of different kinds of addressable storage. This storage may be disjoint in that two storage locations can have the same address, accesses to them being disambiguated either by the instruction used or a flag specifying the currently active storage bank. Optimally allocating declared objects to these storage areas is an active research topic.[3, 16] One implementation[3] distributes the stack over several storage banks.

Coding Guidelines

In resource-constrained environments there can be space and efficiency issues associated with the different kinds of storage durations. These are discussed for each of the storage durations later.

The term *allocated storage* is not commonly used by developers (in the sense of being a noun). The use of the word *allocated* as an adjective is commonly heard. The terms *dynamically allocated* or *allocated on the heap* are commonly used to denote this kind of storage duration. There does not seem to be any worthwhile benefit in trying to educate developers to use the technically correct term in this case.

Usage

In the translated form of this book's benchmark programs 37% of defined objects had static storage duration and 63% had automatic storage duration (objects with allocated storage duration were not included in this count).

Table 449.1: Total number of objects allocated (in thousands), the total amount of storage they occupy (in thousands of bytes), their average size (in bytes) and the high water mark of these values (also in thousands). Adapted from Detlefs, Dosser and Zorn.[6]

Program	Total Objects	Total Bytes	Average Size	Maximum Objects	Maximum Bytes
sis	63,395	15,797,173	249.2	48.5	1,932.2
perl	1,604	34,089	21.3	2.3	116.4
xfig	25	1,852	72.7	19.8	1,129.3
ghost	924	89,782	97.2	26.5	2,129.0
make	23	539	23.0	10.4	208.1
espresso	1,675	107,062	63.9	4.4	280.1
ptc	103	2,386	23.2	102.7	2,385.8
gawk	1,704	67,559	39.6	1.6	41.0
cfrac	522	8001	15.3	1.5	21.4

450 Allocated storage is described in 7.20.3.

Commentary

Allocated storage is not implicitly handled by the implementation. It is controlled by calling library functions.

Other Languages

In some languages handle allocated storage is part of the language, not the library. For instance, C++ contains the **new** operator (where the amount of storage to allocate is calculated by the translator, based on deducing the type of object required). Pascal also contains new, but calls is a *required function*.

---

lifetime of object

The *lifetime* of an object is the portion of program execution during which storage is guaranteed to be reserved for it. 451

Commentary

This defines the term *lifetime*. The storage reserved for an object may exist outside of its guaranteed lifetime. However, this behavior is specific to an implementation and cannot be relied on in a program.

C90

The term *lifetime* was used twice in the C90 Standard, but was not defined by it.

C++

3.8p1 *The lifetime of an object is a runtime property of the object. The lifetime of an object of type T begins when:*

. . .

The lifetime of an object of type T ends when:

The following implies that storage is allocated for an object during its lifetime:

3.7p1 *Storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object.*

Common Implementations

An implementation may be able to deduce that it is only necessary to reserve storage for an object during a subset of the lifetime required by the standard. For instance, in the following example the accesses to the objects loc_1 and loc_2 occur in disjoint portions of program execution. This usage creates an optimization opportunity (having the two objects share the same storage during disjoint intervals of their lifetime).

```
1  void f(void)
2  {
3  int loc_1,
4      loc_2;
5  /*
6   * Portion of program execution that accesses loc_1, but not loc_2.
7   */
8
9  /*
10  * Portion of program execution that accesses loc_2, but not loc_1.
11  */
12 }
```

Reuse of storage is usually preceded by some event; for instance, a function return followed by a call to another function (for automatic storage duration, the storage used by the previous function is likely to be used by different objects in the newly called function). There are many possible algorithms[1, 2, 4, 8, 9, 14, 15] that an implementation can use to manage allocated storage and no firm prediction on reuse can be made about objects having this storage duration.

A study by Bowman, Ratliff, and Whalley[5] optimized the storage lifetimes of the static data used by 15 Unix utilities (using the same storage locations for objects accessed during different time intervals; they all had static storage duration). They were able to make an overage saving of 7.4%. By overlaying data storage with instruction storage (the storage for instructions not being required after their last execution) they achieved an average storage saving of 22.8%.

The issue of where storage is allocated for objects is discussed elsewhere.

object
reserve storage

Coding Guidelines

The lifetime of objects having static or automatic storage durations are easy to deduce from looking at the source code. The lifetime of allocated storage is rarely easy to deduce. Coding techniques to make it easier to demark the lifetime of allocated storage are outside the scope of this book (the Pascal `mark/release` functionality sometimes encouraged developers to develop algorithms to treat heap allocation in a stack-like fashion).

Example

The following coding fault is regularly seen:

```
1  struct list *p;
2    /* ... */
3  free(p);
4  p = p->next;
```

---

452 An object exists, has a constant address,[25)] and retains its last-stored value throughout its lifetime.[26)]

Commentary

At first glance the phrase *constant address* appears to prevent a C implementation from using a garbage collector that moves objects in storage. But what is an address? An implementation could choose to represent object addresses as an index into an area of preallocated storage. This indexed element holds the real address in memory of the object's representation bits. The details of this extra indirection operation is dealt with by the translator (invisible to the developer unless a disassembled listing of the generated code is examined). A garbage collector would only need to update the indexed elements after storage had been compacted, and the program would know nothing about what had happened.

The last-stored value may have occurred as a result of an assignment operator, external factors for an object declared with the **`volatile`** storage-class, or another operator that updates the value held in an object.

C++

There is no requirement specified in the C++ Standard for an object to have a constant address. The requirements that are specified include:

> *Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).* 1.9p10

> *All objects which neither have dynamic storage duration nor are local have static storage duration. The storage for these objects shall last for the duration of the program (3.6.2, 3.6.3).* 3.7.1p1

Common Implementations

In most implementations objects exist in either RAM or ROM. The value of an object whose address is never taken may only ever exist in a processor register; the only way to tell is by looking at a listing of the generated machine code. In all commercial implementations known to your author an object's address has a

direct correspondence to its actual address in storage. There is no indirection via other storage performed by the implementation, although the processor's memory-management unit may perform its own mappings into physical memory.

Coding Guidelines

An implementation that performs garbage collection may have one characteristic that is visible to the developer. The program may appear to stop executing periodically because garbage collection is taking place. There are implementation techniques that perform incremental garbage collection, which avoids this problem to some degree.[11] However, this problem is sufficiently rare that it is considered to be outside the scope of these coding guidelines.

A program running in a sufficiently hostile environment that the last-stored value of an object may be corrupted requires the use of software development techniques that are outside the scope of this book.

Example

The two values output on each iteration of the loop are required to be equal. However, there is no requirement that the values be the same for different iterations of the loop.

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5  for (int loop=0; loop < 10; loop++)
6     {
7     int nested_obj;
8
9     printf("address is=%p", &nested_obj);
10    printf(", and now it is=%p\n", &nested_obj);
11    }
12 }
```

If an object is referred to outside of its lifetime, the behavior is undefined. 453

Commentary

Such a reference is possible through

- the address of a block scope object assigned to a pointer having a greater lifetime, and
- an object allocated by the memory-allocation library functions that has been freed.

C++

The C++ Standard does not unconditionally specify that the behavior is undefined (the cases associated with pointers are discussed in the following C sentence):

3.8p3 *The properties ascribed to objects throughout this International Standard apply for a given object only during its lifetime. [Note: in particular, before the lifetime of an object starts and after its lifetime ends there are significant restrictions on the use of the object, as described below, in 12.6.2 and in 12.7. describe the behavior of objects during the construction and destruction phases.]*

Common Implementations

object 458
lifetime from entry to exit of block

On entry to a function it is common for the total amount of stack storage for that invocation to be allocated. The extent to which the storage allocated to objects, defined in nested blocks, is reused will depend on whether their lifetime is disjoint from objects defined in other nested blocks. In other words, is there an opportunity for a translator to reuse the same storage for different objects?

Most implementations' undefined behavior is to continue to treat the object as if it was still live. The storage location referenced may contain a different object, or even some internal information used by the runtime system. Values read from that location may be different from the last-stored value written into the original object. Stores to that location could affect the values of other objects and the effects of modifying internal, housekeeping information can cause a program to abort abnormally. The situation with allocated storage is much more complex.

Coding Guidelines

When the lifetime of an object ends, nothing usually happens to the last-stored value (or indeed any subsequent ones) held at that location. Programs that access the storage location that held the object, soon after the object's lifetime has ended, often work as expected (such an access can only occur via a pointer dereference; if an identifier denoting a declared object is visible the object it denotes cannot be outside of its lifetime). Accessing an object outside of its lifetime is unlikely to cause the implementation to issue a diagnostic. However, accessing an object outside of its lifetime sometimes does result in unexpected behavior. A coding guideline recommending that "an object shall not be referenced outside of its lifetime." is a special case of the guideline stating that programs shall not contain faults. The general aim of guideline recommendations in this area is to prevent the address of an object being available outside of its lifetime.

object reference outside lifetime

guidelines not faults

Although some implementations provide a mechanism to initialize newly created objects with some unusual value (this often helps to catch uninitialized objects quickly during testing), an equivalent mechanism at the end of an object's lifetime is unknown (to your author).

---

454 The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

pointer indeterminate

Commentary

It is not the object pointed at, but the value of pointers pointing at it that become indeterminate. Once its value becomes indeterminate, the value of the pointer cannot even be read; for instance, compared for equality with another pointer.

An object having a pointer type has an indeterminate value at the start of its lifetime, like any other object (even if that lifetime starts immediately after it was terminated; for instance, an object defined in the block scope of a loop).

461 object initial value indeterminate

C++

The C++ Standard is less restrictive; if does not specify that the value of the pointer becomes indeterminate.

> *Before the lifetime of an object has started but after the storage which the object will occupy has been allocated[34] or, after the lifetime of an object has ended and before the storage which the object occupied is reused or released, any pointer that refers to the storage location where the object will be or was located may be used but only in limited ways. Such a pointer refers to allocated storage (3.7.3.2), and using the pointer as if the pointer were of type* `void*`*, is well-defined. Such a pointer may be dereferenced but the resulting lvalue may only be used in limited ways, as described below. If the object will be or was of a class type with a nontrivial destructor, and the pointer is used as the operand of a* `delete-expression`*, the program has undefined behavior.*

3.8p5

Source developed using a C++ translator may contain pointer accesses that will cause undefined behavior when a program image created by a C implementation is executed.

Other Languages

Languages in the Pascal/Ada family only allow pointers to refer to objects with allocated storage lifetime. These objects can have their storage freed. In this case the same issues as those in C apply.

Common Implementations

Some processors load addresses into special registers (sometimes called address registers; for instance, the Motorola 68000[12]). Loading a value into such an address register may cause checks on its validity as an address to be made by the processor. If the referenced address refers to storage that is no longer available to the program, a trap may be generated.

pointer cause undefined behavior

Coding Guidelines

One way to ensure that pointers never refer to objects whose lifetime has ended is to ensure they are never assigned the address of an object whose lifetime is greater than their own. Scope is a concept that developers are more familiar with than lifetime and a guideline recommendation based on scope is likely to be easier to learn. The applicable recommendations is given elsewhere.

object ??
address assigned

Returning the address of a block scope object, in a **`return`** statement, is a fault. Although other guidelines sometimes recommend against this usage, these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines
not faults

---

static storage duration

An object whose identifier is declared with external or internal linkage, or with the storage-class specifier **`static`** has *static storage duration*. 455

Commentary

This defines the term *static storage duration*. Objects have storage duration, identifiers have linkage. The visibility of an identifier defined with internal linkage may be limited to the block that defined it, but its lifetime extends from program startup to program termination.

Objects declared in block scope can have static storage duration. The **`extern`** or **`static`** storage-class specifiers can occur on declarations in block scope. In the former case it refers to a definition outside of the block. In the latter case it is the definition.

string literal
static storage duration

All file scope objects have static storage duration. String literals also have static storage duration.

C++

The wording in the C++ Standard is not based on linkage and corresponds in many ways to how C developers often deduce the storage duration of objects.

3.7.1p1 *All objects which neither have dynamic storage duration nor are local have static storage duration.*

3.7.1p3 *The keyword* ***`static`*** *can be used to declare a local variable with static storage duration.*

Common Implementations

The total number of bytes of storage required for static storage duration objects is usually written into the program image in translation phase 8. During program startup this amount of storage is requested from the host environment. Objects and literals that have static storage duration are usually placed in a fixed area of memory, which is reserved on program startup. This is possible because the amount of storage needed is known prior to program execution and does not change during execution.

Coding Guidelines

There can be some confusion, in developers' minds, between the keyword **`static`**, static storage duration, and *declared static*— a term commonly used to denote internal linkage. There are even more uses of the keyword **`static`** discussed elsewhere. There is little to be done, apart from being as precise as possible, to reduce the number of possible alternatives when using the word static.

declarator
syntax

Objects in block scope, having static storage duration, retain their last-stored value between invocations of the function that contains their definition. Mistakes are sometimes made on initializing such objects; the developer forgets that initialization via a statement, rather than via an initialization expression in the definition, will be executed every time the function is called. Assigning a value to such an object as its first usage suggests that either static storage duration was not necessary or that there is a fault in the code. While this usage might be flagged by a lint-like tool, neither of them fall within the remit of guideline recommendations.

Example

Here are three different objects, all with static storage duration.

```
1  static int vallu;
2
3  void f(void)
4  {
5  static int valyou = 99;
6
7     {
8     static int valu;
9
10    valu = 21;
11    }
12 }
```

Usage

In the visible form of the `.c` files approximately 5% of occurrences of the keyword **`static`** occurred in block scope.

---

456 Its lifetime is the entire execution of the program and its stored value is initialized only once, prior to program startup.

static storage duration when initialized

Commentary

The storage is allocated and initialized prior to calling the function `main`. A recursive call to `main` does not cause startup initialization to be performed again.

program startup

Other Languages

Java loads modules on an as-needed basis. File scope objects only need be allocated storage when each module is loaded, which may be long after the program execution started.

Common Implementations

Implementations could use the as-if rule to delay creating storage for an object, with static storage duration, until the point of its first access. There were several development environments in the 1980s that used incremental linkage at the translation unit level. These environments were designed to aid the testing and debugging of programs, even if the entire source base was not available.

Coding Guidelines

In environments where storage is limited, developers want to minimize the storage footprint of a program. If some objects with static storage duration are only used during part of a program's execution, more efficient storage utilization schemes may be available; in particular making use of allocated storage.

Use of named objects makes it easier for a translator, or static analysis tool, to detect possible defects in the source code. Use of pointers to objects requires very sophisticated points to analysis just to be able to do the checks performed for named objects (without using sophisticated analysis).

A technique that uses macros to switch between referencing named objects during development and allocated storage during testing and deployment offers a degree of independent checking. If this technique is used, it is important that testing be carried out using the configuration that will ship in the final product. Using named objects does not help with checking the lifetimes of the allocated objects used to replace them. However, discussion of techniques for controlling a program's use of storage is outside the scope of this book.

---

457 An object whose identifier is declared with no linkage and without the storage-class specifier **`static`** has *automatic storage duration*.

automatic storage duration

Commentary

These objects occur in block scope and are commonly known as *local variables*. The storage-class specifier **auto** can also be used in the definition of such objects. Apart from translator test programs, this keyword is rarely used, although some developers use it as a source layout aid.

C++

3.7.2p1 *Local objects explicitly declared* **auto** *or* **register** *or not explicitly declared* **static** *or* **extern** *have automatic storage duration.*

Other Languages

Nearly all languages have some form of automatic storage allocation for objects. Use of a language keyword to indicate this kind of storage allocation is very rare. Cobol does not support any form of automatic storage allocation.

Common Implementations

Storage for objects with automatic storage duration is normally reserved on a stack. This stack is frequently the same one used to pass arguments to functions and to store function return addresses and other housekeeping information associated with a function invocation (see Figure **??**).

Recognizing the frequency with which such automatic storage duration objects are accessed (at least while executing within the function that defines them), many processor instruction sets have special operations, or addressing modes, for accessing storage within a short offset from an address held in a register.

register + offset

The Dynamic C implementation for Rabbit Semiconductor[19] effectively assigns the static storage class to all objects declared with no linkage (this default behavior, which does not support recursive function calls, may be changed using the compiler **#class auto** directive).

Coding Guidelines

The terminology *automatic storage duration* is rarely used by developers. Common practice is to use the term *block scope* to refer to such objects. The special case of block scope objects having static storage duration is called out in the infrequent cases it occurs.

---

object lifetime from entry to exit of block

For such an object that does not have a variable length array type, its lifetime extends from entry into the block with which it is associated until execution of that block ends in any way. 458

Commentary

identifier scope begins
block scope terminates

While the lifetime of an object may start on entry into a block, its scope does not start until the completion of the declarator that defines it (an objects scope ends at the same place as its lifetime).

C++

3.7.2p1 *The storage for these objects lasts until the block in which they are created exits.*

5.2.2p4 *The lifetime of a parameter ends when the function in which it is defined returns.*

6.7p2 *Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).*

Which is a different way of phrasing 3.7.2p1.

3.8p1

The lifetime of an object of type T begins when:

— storage with the proper alignment and size for type T is obtained, and

. . .

The lifetime of an object of type T ends when:

— the storage which the object occupies is reused or released.

The C++ Standard does not appear to completely specify when the lifetime of objects created on entry into a block begins.

Other Languages

All arrays in Java, Awk, Perl, and Snobol 4 have their length decided during program execution (in the sense that the specified size is not used during translation even if it is a constant expression). As such, the lifetime of array in these languages does not start until the declaration, or statement, that contains them is executed.

Common Implementations

Most implementations allocate the maximum amount of storage required, taking into account definitions within nested blocks, on function entry. Such a strategy reduces to zero any storage allocation overhead associated with entering and leaving a block, since it is known that once the function is entered the storage requirements for all the blocks it contains are satisfied. Allocating and deallocating storage on a block-by-block basis is usually considered an unnecessary overhead and is rarely implemented. In:

```
1  void f(void)
2  {               /* block 1 */
3  int loc_1;
4
5     {            /* block 2 */
6     long loc_2;
7     }
8
9     {            /* block 3 */
10    double loc_3;
11    }
12 }
```

the storage allocated for `loc_2` and `loc_3` is likely to overlap. This is possible because the blocks containing their definitions are not nested within each other, and their lifetimes are disjoint. On entry to `f` the total amount of storage required for developer-defined automatic objects (assuming `sizeof(long) <= sizeof(double)`) is `sizeof(int)+padding_for_double_alignment+sizeof(double)`.

Coding Guidelines

Where storage is limited, defining objects in the closest surrounding block containing accesses to them, can reduce the maximum storage requirements (because objects defined with disjoint lifetimes can share storage space). It is safer to let the translator perform the housekeeping needed to handle such shared storage than to try to do it manually (by using of the same objects for disjoint purposes). The issues surrounding the uses to which an object is put are discussed elsewhere. The issues involved in deciding which block an identifier should be defined in, if it is only referenced within a nested block, are also discussed elsewhere.

?? object
used in a single role
identifier definition
close to usage

Example

```
1  int glob;
2  int *pi = &glob;
3
4  void f(void)
5  { /* Lifetime of loc starts here. */
```

```
6
7   block_start:;
8
9   *pi++; /* The identifier loc is not visible here. */
10
11  if (*pi == 1)
12     goto skip_definition; /* Otherwise we always execute initializer. */
13
14  int loc = 1;
15
16  skip_definition:;
17
18  pi=&loc;
19
20  if (loc != 7)
21     goto block_start;
22
23  /* Lifetime of loc ends here. */ }
```

(Entering an enclosed block or calling a function suspends, but does not end, execution of the current block.) 459

Commentary

Execution of a block ends when control flow within that block transfers execution to a block at the same or lesser block nesting, or causes execution of the function in which it is contained to terminate (i.e., by executing a **`return`** statement, or calling one of the `longjmp`, `exit`, or `abort` functions).

C is said to be a *block structured* language.

C++

1.9p10 *Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function or receipt of a signal).*

3.7.2p1 *The storage for these objects lasts until the block in which they are created exits.*

Other Languages

This behavior is common to block structured languages.

Coding Guidelines

It would be incorrect to assume that objects defined with the **`volatile`** qualifier can only be modified by the implementation while the block that defines them is being executed. Such objects can be modified at any point in their lifetime.

---

block entered recursively

If the block is entered recursively, a new instance of the object is created each time. 460

Commentary

function call recursive

This can only happen through a recursive call to the function containing the block. A jump back to the beginning of the block, using a **`goto`** statement, is not a recursive invocation of that block.

C90

The C90 Standard did not point this fact out.

C++

As pointed out elsewhere, the C++ Standard does not explicitly specify when storage for such objects is created. However, recursive instances of block scope declarations are supported.

458 object lifetime from entry to exit of block

> *Recursive calls are permitted, except to the function named* `main` *(3.6.1).* 5.2.2p9

> *Variables with automatic storage duration (3.7.2) are initialized each time their* `declaration-statement` *is executed. Variables with automatic storage duration declared in the block are destroyed on exit from the block (6.6).* 6.7p2

Other Languages

This behavior is common to block structured languages. Recursion was not required in the earlier versions of the Fortran Standard, although some implementations provided it.

Common Implementations

There are some freestanding implementations where storage is limited and recursive function calls are not supported (such implementations are not conforming). The problem is not usually one of code generation, but the storage optimizations performed by the linker. To minimize storage usage in memory limited environments linkers build program call graphs to deduce which objects can have their storage overlaid. The storage optimization algorithms do not terminate if there are loops in the call graph (a recursive call would create such a loop). Thus, recursion is not supported because programs containing it would never get past translation phase 8.

call graph

Allocating storage for all objects defined in a function, on a per function invocation basis, may cause inefficient use of storage when recursive invocations occur. In practice both recursive invocations and objects defined in nested blocks are rare.

---

461 The initial value of the object is indeterminate.

object initial value indeterminate

Commentary

This statement applies to all object declarations having no linkage, whether they have initializers or not (it is explicitly stated for objects having a VLA type elsewhere).

466 object initial VLA value indeterminate

C++

> *If no initializer is specified for an object, and the object is of (possibly cv-qualified) non-POD class type (or array thereof), the object shall be default-initialized; if the object is of const-qualified type, the underlying class type shall have a user-declared default constructor.*
>
> *Otherwise, if no initializer is specified for an object, the object and its subobjects, if any, have an indeterminate initial value[90]; if the object or any of its subobjects are of const-qualified type, the program is ill-formed.* 8.5p9

C does not have constructors. So a const-qualified object definition, with a structure or union type, would be ill-formed in C++.

```
1  struct T {int i;};
2
3  void f(void)
4  {
5  const struct T loc; /* very suspicious, but conforming */
6                      // Ill-formed
7  }
```

Other Languages

This behavior is common to nearly every block scoped language. Some languages (e.g., awk) provide an initial value for all objects (usually zero, or the space character).

Common Implementations

Some implementations assign a zero value to automatic and allocated storage when it is created. They do this to increase the likelihood that programs containing accesses to uninitialized objects will work as intended. They are *application user friendly* by helping to protect against errors in the source code. (Many objects are initially set to zero by developers and this implicit implementation value assignment mimics what is likely to be the behavior intended by the author of the code.)

Some implementations (e.g., the Diab Data compiler[7] supports the `-Xinit-locals-mask` option) implicitly assign some large value, or a trap representation, to freshly allocated storage. The intent is to be *developer friendly* by helping to detect faults, created by use of uninitialized objects, as quickly as possible. Assigning an unusual value is likely to have the effect of causing the reads from uninitialized objects to have an unintended effect and to be quickly detected. The Burroughs B6700 (a precursor of the Unisys A Series[18]) initialized its 48-bit words to 0xbadbadbadbad. The value 0xdeadbeef is used in a number of environments supported by IBM.

Coding Guidelines

It is surprising how often uninitialized objects contain values that result in program execution producing reasonable results. Having the implementation implicitly initialize objects to some *unfriendly* value helps to track down these kinds of faults much more quickly and helps to prevent reasonable results from hiding latent problems in the source code.

---

initialization performed every time declaration reached

If an initialization is specified for the object, it is performed each time the declaration is reached in the execution of the block; 462

Commentary

object initializer evaluated when

The sequence of operations is specified in more detail elsewhere in the C Standard.

If flow of control jumps over the declaration, for instance by use of a **goto** statement, the initialization is not performed; however, the lifetime of the object is not affected (the storage will already have been allocated on entry into the block).

object 458 lifetime from entry to exit of block

Because initializations are performed during program execution, it is possible for the evaluation of a floating constant to cause an exception to be raised. This can occur even if the floating constant appears to have the same type as the object it is being assigned to. An implementation is at liberty to evaluate the initialization expression in a wider format, which will then need to be converted to the object type. It is the conversion from any wider format to the type of the object type that may raise the exception.

FLT_EVAL_METHOD

```
1  #include <math.h>
2
3  void f(void)
4  {
5  static float w = 1.1e75; /* Performed at translation time, no exception raised. */
6  /*
7   * The following may all require conversions at execution time.
8   * Therefore they can all raise exceptions.
9   */
10 float x_1 = 1.1e75;
11 double x_2 = 1.1e75;
12 float x_3 = 1.1e75f;
13
14 /*
15  * The following do not require any narrowing conversions and cannot raise exceptions.
16  */
17 long double x_4 = 1.1e75;
```

```
18  double_t x_5 = 1.1e75;
19
20  /*
21   * For constant expressions we have (the final value of y and z is undefined)...
22   */
23  static double y = 0.0/0.0;  /* Performed at translation time, no exception raised. */
24  auto double z = 0.0/0.0;    /* Performed at execution time, may raise exception.   */
25  }
```

C90

If an initialization is specified for the value stored in the object, it is performed on each normal entry, but not if the block is entered by a jump to a labeled statement.

Support for mixing statements and declarations is new in C99. The change in wording is designed to ensure that the semantics of existing C90 programs is unchanged by this enhanced functionality.

Other Languages

Some languages do not allow object definitions to contain an initializer. Those that do usually follow the same initialization rules as C.

Common Implementations

For objects having a scalar type the machine code generated for the initializer will probably look identical to that generated for an assignment statement. For derived types implementations have all the information needed to generate higher-quality code. For instance, for array types, the case of all elements having a zero value is usually a special case and a machine code loop is generated to handle it.

Example

```
1   void f(int x)
2   {
3   switch(x)
4      {
5      int i=33; /* This initialization never occurs. */
6
7      case 1: x--;
8              break;
9      }
10  }
```

Usage

Usage information on initializers is given elsewhere.

object value indeterminate

---

463 otherwise, the value becomes indeterminate each time the declaration is reached.

object indeterminate each time declaration reached

Commentary

An indeterminate value is stored in the object each time the declaration is reached in the order of execution.

object initializer evaluated when

Other Languages

Most languages follow the model of giving an object an indeterminate value when it is first defined.

Common Implementations

If no other objects are assigned the same storage location, it is very likely that the last-stored value will be available in the object every time its declaration is reached.

Example

```
1  extern int glob;
2
3  _Bool f(void)
4  {
5  for (int i=0; i<5; i++)
6     {
7     int loc;
8
9     if (i > 0)
10       loc -= i; /* loc always has an indeterminate value here. */
11    else
12       loc=23;
13    }
14
15 goto do_work;
16    {
17    start_block:;
18    int count;
19
20    return (count == glob); /* count always has an indeterminate value here. */
21    /*
22     * The above return statement exhibits undefined behavior,
23     * because of the access to count.
24     */
25    do_work:;
26    count = glob % 4;
27
28    goto start_block;
29    }
30 }
```

VLA lifetime starts/ends

For such an object that does have a variable length array type, its lifetime extends from the declaration of the object until execution of the program leaves the scope of the declaration.[27)] 464

Commentary

The lifetime of a VLA starts at its point of declaration, not at the point of entry into the block containing its definition. This means it is possible to control the number of elements in the VLA through side effects within the block containing its definition. The intent of the Committee was for it to be possible to implement VLAs using the same stack used to allocate storage for local objects. Although their size varies, the requirements are such that it is possible to allocate storage for VLAs in a stack-like fashion. C does not define any out-of-storage signals and is silent on the behavior if the implementation cannot satisfy the requested amount of storage.

C90

Support for variable length arrays is new in C99.

C++

C++ does not support variable length arrays in the C99 sense; however, it does support containers:

23.1p1 *Containers are objects that store other objects. They control allocation and deallocation of these objects through constructors, destructors, insert and erase operations.*

The techniques involved, templates, are completely different from any available in C and are not discussed further here.

Other Languages

The lifetime of arrays in some languages (e.g., Java, Awk, Perl, and Snobol 4) does not start until the declaration, or statement, that contains them is executed. An array definition in Java does not allocate storage for the array elements; it only allocates storage for a reference to the object. The storage for the array elements is created by the execution of an *array access expression*. This can occur at any point where the identifier denoting the declared array is visible. Fortran and Pascal allow variable-size arrays to be passed as arguments. However, this is a subset of the functionality involved in allowing the number of elements in an array definition to be decided during program execution. In PL/1 the storage for arrays whose size is computed during program execution is not allocated at the point of definition, but within the executable statements via the use of the **allocate** statement.

```
declare
     (k, l) fixed bin,
     a dim (k) char (l) controlled;

k = 10;
l = 10;
allocate a;
free a;
```

Common Implementations

Because the size of an object having VLA type is not known at translation time, the storage has to be allocated at execution time. A commonly used technique involves something called a *descriptor*. Storage for this descriptor, one for each object having VLA type, is allocated in the same way as locally defined objects having other types, during translation. The descriptor contains one or more members that will hold information on the actual storage allocated and for multidimensional arrays the number of elements in each dimension. When the definition of an object having a VLA type is encountered during program execution, storage is allocated for it (often on the stack), and the descriptor members are filled in.

```
extern void G(void);

void f(int n)
{
long l;
char a[n];
float f_l;

G();
double d[11][n+8];
int i_l;
/* ... */
}
```

Once execution leaves the scope of the object definition, having a VLA type, its lifetime terminates and the allocated storage can be freed up. This involves the implementation keeping track of all constructs that can cause this to occur (e.g., **goto** statements, as well as normal block termination).

One technique for simplifying the deallocation of VLA stack storage is to save the current value of the stack pointer on entry into a block containing VLA object definitions and to restore this value when the block terminates. There are situations where this technique might not be considered applicable (e.g., when a **goto** statement jumps to before a VLA object definition in the same block, an implementation either has to perform deallocation just prior to the **goto** statement or accept additional stack growth).

Coding Guidelines

For objects that don't have a VLA type, jumping over their definition may omit any explicit initialization, but storage for that object will still have been allocated. If the object has a VLA type the storage is only allocated

458 object lifetime from entry to exit of block

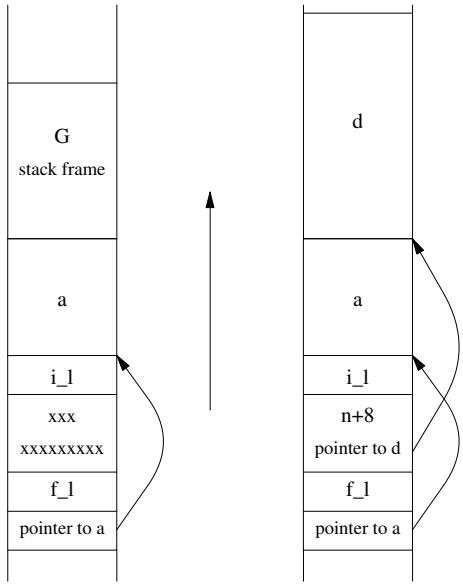

Figure 464.1: Storage for objects not having VLA type is allocated on block entry, plus storage for a descriptor for each object having VLA type. By the time G has been called, the declaration for a has been reached and storage allocated for it. After G returns, the declaration for d is reached and is storage allocated for it. The descriptor for d needs to include a count of the number of elements in one of the array dimensions. This value is needed for index calculations and is not known at translation time. No such index calculations need to be made for a.

if the definition is executed.

It is not expected that the use of VLA types will cause a change in usage patterns of the **goto** statement. A guideline recommendation dealing with this situation is not considered to be worthwhile.

Example

```
1  extern int n;
2  extern char gc;
3  extern char *pc = &gc;
4
5  void f(void)
6  {
7  block_start:;
8  /*
9   * At this point the lifetime of ca has not yet started.  Jumping back to
10  * this point, from a point where the lifetime has started, will terminate
11  * the lifetime.
12  */
13 n++;
14
15 /*
16  * First time around the loop pc points at storage allocated for gc.
17  * On second and subsequent iterations pc will have been pointing at an
18  * object whose lifetime has terminated (giving it an indeterminate value).
19  */
20 pc[0]='z';
21
22 char ca[n]; /* Lifetime of ca starts here. */
23 pc=ca;
24
25 if (n < 11)
```

```
26      goto block_start;
27  }
```

---

465 If the scope is entered recursively, a new instance of the object is created each time.

object
new instance
for recursion

Commentary

A scope can only be entered recursively through a recursive function call. This behavior is the same as for objects that don't have a VLA type. Functions containing VLAs are reentrant, just like functions containing any other object type.

460 block
entered recursively

The functions in the standard library are not guaranteed to be reentrant.

Other Languages

This behavior is common to all block scoped languages.

Common Implementations

In practice few function definitions are used in a way that requires their implementation to be reentrant. Some implementations[17] assume that functions need not be reentrant unless explicitly specified as such (e.g., by using a keyword such as **`reentrant`**).

---

466 The initial value of the object is indeterminate.

object
initial VLA value
indeterminate

Commentary

Objects having a VLA type are no different from objects having any other type.

461 object
initial value
indeterminate

Common Implementations

The usage patterns of objects having a VLA type are not known. Whether they are less likely to have their values from a previous lifetime in a later lifetime than objects having other types is not known.

---

467 **Forward references:** statements (6.8), function calls (6.5.2.2), declarators (6.7.5), array declarators (6.7.5.2), initialization (6.7.8).

---

468 25) The term "constant address" means that two pointers to the object constructed at possibly different times will compare equal.

footnote
25

Commentary

The standard does not specify how addresses are to be represented in a program; it only specifies the results of operations on them. In between their construction and being compared, it is even possible that they are written out and read back in again.

C++

The C++ Standard is silent on this issue.

Other Languages

This statement is true of nearly all classes of languages, although some don't support the construction of pointers to objects. Functional languages (often used when formally proving properties of programs) never permit two pointers to refer to the same object. Assignment of pointers always involves making a copy of the pointed-to object (and returning a pointer to it).

Common Implementations

In some implementations the address of an object is its actual address in the processes (executing the program) address space. With memory-mapping hardware (now becoming common in high-end freestanding environments) it is unlikely to be the same as its physical address. Its logical address may remain constant, but its physical address could well change during the execution of the program.

The address may be different during two different executions of the same program. 469

Commentary

It is possible to write a pointer out to a file using the %p conversion specifier during one execution of the program and read it back in during a subsequent execution of the program. While the address read back, during the same execution of the program, will refer to the same object (assuming the object lifetime has not ended); however, during a different execution of the program the address is not guaranteed to refer be the same object (or storage allocated for any object).

Addresses might be said to having two kinds of representation details associated with them. There is the bit pattern of the value representation and there is the relative location of one address in relation to another address. Some applications make use of the relationship between addresses for their own internal algorithms. For instance, some garbage collectors for Lisp interpreters depend on low addresses being used for allocated storage. Low address are required because the garbage collector use higher order bits within the address value to indicate certain properties (assuming they can be zeroed out when used in a pointer dereference).

object
reserve storage

C++

The C++ Standard does not go into this level of detail.

Other Languages

Most high-level languages do not make visible, to the developer, the level of implementation detail specified in the C Standard.

Common Implementations

Programs running in hosted environments that use a memory-management unit to map logical to physical addresses are likely to use the same logical addresses, to hold the same objects, every time they are executed. It is only when other programs occupy some of the storage visible to a program that the address of its objects is likely to vary.

footnote 26

26) In the case of a volatile object, the last store need not be explicit in the program. 470

Commentary

volatile
last-stored value

The fact that a volatile object may be mapped to an I/O port, not a storage location, does not alter its lifetime. As far as a program is concerned, the lifetime of an object having a given storage-class is defined by the C Standard.

C++

7.1.5.1p8 *[Note: **volatile** is a hint to the implementation to avoid aggressive optimization involving the object because the value of the object might be changed by means undetectable by an implementation. See 1.9 for detailed semantics. In general, the semantics of **volatile** are intended to be the same in C++ as they are in C.]*

footnote 27

27) Leaving the innermost block containing the declaration, or jumping to a point in that block or an embedded block prior to the declaration, leaves the scope of the declaration. 471

Commentary

The scope of the declaration is the region of program text in which the identifier is visible using a top-down, left-to-right parse of the source code.

C90

Support for VLAs is new in C99.

Common Implementations

An implementation is required to track all these possibilities. It can choose not to free-up allocated storage in some cases, perhaps because it can deduce that the same amount of storage will be allocated when the definition is next executed. There is little practical experience with the implementation and VLA types at the moment. The common cases can be guessed at, but are not known with certainty.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1985.
2. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architecture*. Morgan Kaufmann Publishers, 2002.
3. O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, 2002.
4. R. Barua. *Maps: A Compiler-Managed Memory System for Software-Exposed Architectures*. PhD thesis, M.I.T., Jan. 2000.
5. R. L. Bowman, E. J. Ratliff, and D. B. Whalley. Decreasing process memory requirements by overlapping program portions. In *Proceedings of the Hawaii International Conference on System Sciences*, pages 115–124, Jan. 1998.
6. D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. Technical Report CU-CS-665-93, University of Colorado at Boulder, Aug. 1993.
7. Diab Data. *D-CC & D-C++ Compiler Suites User's Guide*. Diab Data, Inc, www.ddi.com, 4.3 edition, June 1999.
8. E. Eckstein and A. Krall. Minimizing cost of local variables access for DSP-processors. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES 99)*, volume 34.7 of *ACM SIGPLAN Notices*, pages 20–27. ACM Press, May 5 1999.
9. S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.
10. IAR Systems. *PICmicro C Compiler: Programming Guide*, iccpic-1 edition, 1998.
11. R. Jones and R. Lims. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Addison–Wesley, 1996.
12. Motorola, Inc. *MOTOROLA M68000 Family Programmer's Reference Manual*. Motorola, Inc, 1992.
13. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.
14. A. Rao. Compiler optimizations for storage assignment on embedded DSPs. Thesis (m.s.), University of Cincinnati, Oct. 1998.
15. E. L. Robertson. Code generation and storage allocation for machines with span-dependent instructions. *ACM Transactions on Programming Languages and Systems*, 1(1):71–83, 1979.
16. J. Sjödin and C. von Platen. Storage allocation for embedded processors. In *Proceedings of CASES'01*, pages 15–23, Nov. 2001.
17. Texas Instruments. *TMS370 and TMS370C8 8-Bit Microcontroller Family Optimizing C Compiler Users' Guide*. Texas Instruments, spnu022c edition, Apr. 1996.
18. Unisys Corporation. *Architecture MCP/AS (Extended)*. Unisys Corporation, 3950 8932-100 edition, 1994.
19. Z-World. *Dynamic C User's Manual*. Z-World, Inc, Davis, CA, USA, 019-0071.020218-p edition, 1999.