The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

3.2

alignment

alignment 39

requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

Commentary

In an ideal world there would be no alignment requirements. In practice the designers of some processors have placed restrictions on the fetching of variously sized objects from storage. The underlying reason for these restrictions is to simplify (reduce cost) and improve the performance of the processor. Some processors do not have alignment requirements, but may access storage more quickly if the object is aligned on a particular address boundary.

additive operators pointer to object

pointer to void same representation and alignment as

common initial sequence

The requirement that a pointer to an object behave the same as a pointer to an array of that object's type forces the requirement that `sizeof(T)` be a multiple of the alignment of `T`. If two objects, having different types, have the same alignment requirements then their addresses will be located on the same multiple of a byte address. Objects of character type have the least restrictive alignment requirements, compared to objects of other types. Alignment and padding are also behind the assumptions that need to be made for the common initial sequence concept to work.

The extent to which alignment causes implementation-defined or undefined behavior is called out for each applicable construct. Various issues relating to alignment were the subject of DR #074.

C++

3.9p5 *Object types have alignment requirements (3.9.1, 3.9.2). The alignment of a complete object type is an implementation-defined integer value representing a number of bytes; an object is allocated at an address that meets the alignment requirements of its object type.*

There is no requirement on a C implementation to document its alignment requirements.

Other Languages

Most languages hide such details from their users.

Common Implementations

alignment addressable storage unit

Alignment is a very important issue for implementations. Internally it usually involves trade-offs between storage and runtime efficiency. Externally it is necessary to deal with parameter passing interfaces and potentially the layout of structure members. The C language is fortunate in that many system interfaces are specified in terms of C. This means that other languages have to interface to its way of doing things, not the other way around.

Optimal code generation for modern processors requires implementations to consider alignment issues associated with cache lines. Techniques for finding and using the optimal memory alignment for the objects used in a program is an active research area.[3]

Motorola 56000

The Motorola 56000[4] allows its modulo arithmetic mode to be applied to pointer arithmetic operations (used to efficiently implement a circular buffer; the wraparound effectively built into the access instruction). The one requirement is that the storage be aligned on an address boundary that is a power of two greater than or equal to the buffer size. So an array of 10 characters needs to be placed on a 16-byte boundary, while an array of 100 characters would need to be on a 128-byte boundary (assuming they are to be treated as circular buffers). Translators for such processors often provide an additional type specifier to enable developers to indicate this usage.

The Intel XScale architecture[2] has data cache lines that start on a 32-byte address boundary. Arranging for aggregate objects, particularly arrays, to have such an alignment will help minimize the number of stalls while the processor waits for data, and it also enables optimal use of the prefetch instruction. The Intel x86 architecture has no alignment requirement, but can load multibyte objects more quickly if appropriately aligned. RISC processors tend to have strict alignment requirements, often requiring that scalar objects be aligned on a byte boundary that is a multiple of their size.

Does the storage class or name space of an identifier affect its alignment?

Many implementations make no attempt to save storage, for objects having static storage duration, by packing them close together. It is often simpler to give them all the same, worst-case, alignment. For hosted implementations potential savings (e.g., storage saved, time to access) may be small; there don't tend to be significant numbers of scalar objects with static storage duration. However, in those cases where freestanding environments have limited storage availability, vendors often go to great lengths to make savings. The alignment of objects having static storage duration is sometimes controlled by the linker, which may need to consider more systemwide requirements than a C translator (e.g., handling other languages or program image restrictions). For example, the C translator may specify that all objects start on an even address boundary; objects with automatic storage duration being placed on the next available even address after the previous object, but the linker using the next eighth-byte address boundary for objects. linkers

A host's parameter-passing mechanism may have a minimum alignment requirement, for instance at least the alignment of **int**, and stricter alignment for types wider than an **int**. Minimizing the padding between different objects with automatic storage duration reduces the stack overhead for the function that defines them. It also helps to ensure they all fit within any short addressing modes available within the instruction set of the host processor. Unlike objects with static storage duration, most translators attempt to use an alignment that is appropriate to the type of the object.

The `malloc` function does not know the effective type assigned to the storage it allocates. It has to make worst-case assumptions and allocate storage based on the strictest alignment requirements.

The members of a structure type are another context where alignment requirements can differ from the alignment of objects having the same type in non-member contexts. member alignment

Most processor instructions operate on objects having a scalar type and any alignment requirements usually only apply to scalar types. However, some processors contain instructions that operate on objects that are essentially derived types and these also have alignment requirements (e.g., the Pentium streaming SIMD instructions[1]). The multiples that occur in alignment requirements are almost always a power of two. On some processors the alignment multiple of a scalar type is the same as its size, in bytes. derived type

Alignment requirements are not always constant during translation. Several implementations provide **#pragma** directives that control the alignment used by translators. A common vendor extension for controlling the alignment of structure members is the **#pack** preprocessor directive. `gcc` supports several extensions for getting and setting alignment information.

- The keyword **`__alignof__`**, which returns the alignment requirement of its argument, for instance, `__alignof__ (double)` might return 8.
- The keyword **`__attribute__`** can be used to specify an object's alignment at the point of definition:

```
int x __attribute__((aligned(16))) = 0;
```

Coding Guidelines

Alignment is an issue that can affect program resource requirements, program interoperability, and source code portability.

Alignment can affect the size of structure types. Two adjacent members declared with different scalar types may have padding inserted between them. Ordering members such that those with the same type are adjacent to each other can eliminate this padding (because the alignment of a scalar type is often a multiple of its size). If many instances of an object having a particular structure type are created a large amount of storage may be consumed and developers may consider it to be worthwhile investigating ways of making savings; for instance, by changing the default alignment settings, or by using a vendor-supplied **#pragma**.[5] If alignment 39 alignment is changed for a subset of structures there is always the danger that a declaration in one translation unit will have an alignment that differs from the others. Adhering to the guideline recommendation on having a single point of declaration reduces the possibility of this occurring.

identifier ?? declared in one file

Some applications chose to store the contents of objects having structure types, in binary form, in files as a form of data storage. This creates a dependency on an implementation and the storage layout it uses for types. Other programs that access these data files, or even the same program after retranslation, need to ensure that the same structure types have the same storage layout. This is not usually a problem when using the same translator targeting the same host. However, a different translator, or a different host, may do things differently. The design decisions behind the choice of data file formats is outside the scope of these coding guidelines.

storage layout

Programs that cast pointers to point at different scalar types are making use of alignment requirements information, or lack of them. Porting a program which makes use of such casts to a processor with stricter alignment requirements, is likely to result in some form of signal being raised (most processors raise some form of exception when an attempt is made to access objects on misaligned addresses). The cost of modifying existing code to work on such a processor is one of the factors that should have been taken into account when considering the original decision to allow a guideline deviation permitting the use of such casts.

pointer to void converted to/from

Developers sometimes write programs that rely on different types sharing the same alignment requirements, e.g., **`int`** and **`long`**. There are cases where the standard guarantees identical alignments, but in general there are no such guarantees. Such usage is making use of representation information and is covered by a guideline recommendation.

alignment pointer to structures

representation information using ??

Example

Do all the following objects have the same alignment?

```
#include <stdlib.h>
/*
 * #pragma pack
 */

struct S_1 {
            unsigned char mem_1;
            int mem_2;
            float mem_3;
           };

static unsigned char file_1;
static int           file_2;
static float         file_3;

void f(unsigned char param_1,
       int           param_2,
       float         param_3)
{
unsigned char        local_1, *p_uc;
int                  local_2, *p_i;
float                local_3, *p_f;

p_uc = (unsigned char *)malloc(sizeof(unsigned char));
p_i = (int *)malloc(sizeof(int));
p_f = (float *)malloc(sizeof(float));
}
```

No, they need not, although implementations often chose to use the same alignment requirements for each scalar type, independent of the context in which it occurs. The `malloc` function cannot know the use to which the storage it returns will be put, so it has to make worst-case assumptions.

```
/*
 * Declare a union containing all basic types and pointers to them.
 */
union align_u {
        char            c, *cp;
        short                h, *hp;
        int             i, *ip;
        long            l, *lp;
        long long       ll, *llp;
        float           f, *fp;
        double          d, *dp;
        long double     ld, *ldp;
        void           *vp;
/*
 * Pointer-to function.  The standard does not define any generic
 * pointer types, like it does for pointer to void.  The wording
 * only requires the ability to convert.  There is no requirement
 * to be able to call the converted function pointer.
 */
        void          (*fv)(void);
        void          (*fo)();
        void          (*fe)(int, ...);
        };
/*
 * In the following structure type, the first member has type char.
 * The second member, a union type, will be aligned at least to the
 * strictest alignment of its contained types.  Assume that there
 * is no additional padding at the end of the structure declaration
 * than in the union.  Then we can calculate the strictest alignment
 * required by any object type (well at least those used to
 * define members of the union).
 */
struct align_s {
        char            c;
        union align_u  u;
        };

#define _ALIGN_  (sizeof(struct _align_s) - sizeof(union _align_u))
```

References

1. Intel. *Data Alignment and Programming Issues for the Streaming SIMD Extensions with the Intel C/C++ Compiler*. Intel Corporation, 1.1 edition, Jan. 1999.
2. Intel. *Intel XScale Microarchitecture Programmers Reference Manual*. Intel, Inc, 2001.
3. S. Larsen, E. Witchel, and S. Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Report MIT-LCS-TM-621, MIT, USA, Nov. 2001.
4. Motorola, Inc. *DSP5600 24-bit Digital Signal Processor Family Manual*. Motorola, Inc, Austin, TX, USA, dsp56kfamum/ad edition, 1995.
5. Sun. *C User's Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.