C Basics 15: Miscellaneous

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing
10. Multidimensional Arrays

C Basics 15: Miscellaneous

11. Dynamic and Static Memory
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. **Miscellaneous**
    - **alignment**
    - **padding and packing in structs**
    - **struct bit-fields**

Alignment

A memory address is **n-byte aligned** when the address is a *multiple of n bytes* (where n is a power of 2).

(**Virtual and physical memory addresses** on modern CPUs/OSs are in terms of *bytes*, i.e., bytes are the smallest addressible memory units.)

An *n-byte data object* is thus said to be **aligned** if its memory block starts at an address that is *n-byte aligned*.

Otherwise, the object is said to be **misaligned**.

E.g., one-byte objects (e.g., `char`'s) are always aligned, two-byte objects are aligned only if their first byte is in an even-numbered address (0, 2, 4,...), four-byte objects are aligned if their first byte is in an address divisible by 4 (0, 4, 8,...), etc.

Alignment (contd.)

A **word** is the basic number of bits/bytes of memory that a CPU can access and operate on as a unit.

Modern CPUs generally have word sizes of 8, 16, 32, or 64 bits (1, 2, 3, or 4 *bytes*), with 32 and 64 bits now most common (for all but embedded systems).

CPUs access memory a word at a time, but for many CPUs, the words must be *aligned*.

E.g., 32-bit (four-byte) words must start at (byte) addresses that are multiples of 4.

If data objects are *aligned* (and no larger than word size), the CPU can access/operate on the units in a *single CPU cycle*.

Alignment (contd.)

When data objects are *misaligned*, however, the CPU may either not be able to access them or may require multiple cycles to access them (e.g., fetching two words each of which contain part of the object),

The ability of CPU architectures to access/operate on misaligned data, and the speed penalty for operating on such data when possible, varies across CPU architectures.

Because of this, C compilers such as GCC by default generate code that results in all data objects (variables, struct members, etc.) being properly aligned.

(Still, when writing C code that must be portable across 32 and 64-bit x86 as well as ARM, etc., alignment can be an issue.)

Padding

How does a C compiler ensure that data objects are aligned?

In one of two ways:

- adding **padding** bytes between objects
- *reordering* objects (where allowed by C standard)

The easiest way to see padding in action is with *structs*
(C standard does not allow struct members to be reordered).

First, be aware that C compilers generally:

- struct instance will have the alignment of its *widest scalar member* (to ensure that all the members are self-aligned)
- there is no *leading padding*
- *trailing padding* may be added to ensure following object is aligned as struct (**stride address**)

Padding and Structs

Sizes of some C types on 64-bit Linux on x86_64:

- `char`: 1 byte
- `short`: 2 bytes
- `int`: 4 bytes
- `void*`: 8 bytes

An n-byte aligned address will have a minimum of $log_2(n)$ least-significant zeros when expressed in binary.

Addresses are generally given in hex, so only the *least-significant hex digit* is relevant to alignment.

E.g., for 4-byte alignment is would have to be 0, 4, 8, or C (12).

Padding and Structs

Consider the following code:

```
struct test1{
  int x1;
  short x2;
  int x3;
};

int v0 = 0;
struct test1 v1 = {1,2,3};
int v2 = 0;

printf("v0:%p v1:%p x1:%p x2:%p x3:%p size:%ld v2:%p\n",
       &v0,&v1,&v1.x1,&v1.x2,&v1.x3,sizeof(v1),&v2);
```

Output:

```
v0:0x7ffcdf05008c  //v0 4-byte aligned, immediately after struct x3 member
v1:0x7ffcdf050080  //struct start, 4-byte aligned
x1:0x7ffcdf050080  //x1 4-byte aligned, same as start of struct
x2:0x7ffcdf050084  //x2 automatically 4-byte aligned, but only 2-byte required
x3:0x7ffcdf050088  //x3 4-byte aligned, 2 bytes of padding before required
size:12            //struct 12 bytes instead of required 10, due to padding
v2:0x7ffcdf05007c  //v2 4-byte aligned, immediately before struct
```

Struct Packing

The C standard does not allow a compiler to reorder the members of a struct (since the order may required for some reason as in an OS device driver).

However, a program's developer is obviously free to redesign structs so as to minimize padding.

This is known as **struct**(ure) **packing**.

There are two equally valid approaches for minimizing padding in structs:

- order members from smallest to largest
- order members from largest to smallest

Struct Packing (contd.)

While struct packing is not critical in most applications on modern systems, programs that create *many thousands* of struct instances can sometimes significantly improve their memory performance.

On systems with limited RAM, such as many *embedded systems*, struct packing could have a significant impact on memory use (and so performance) or even could be the only way to allow a program to run.

Bit-Fields

Bit-fields (also *bitfields* or *bit fields*) are another approach for packing more struct members into a given number of bytes.

Consider a stuct that contains a number of (binary) **flags**:

```
struct flags {
  int flag1;
  int flag2;
  ...
  int flag8;
};
```

This struct requires $8 \times 4 = 32$ bytes to store only *8 bits* (one byte) of information.

Thus, it wastes 31 bytes per struct instance.

Bit-Fields (contd.)

C bit-fields allow the *number of bits* that should be used for `_Bool`, `signed int`, or `unsigned int` struct members to be specified.

A bit-field is specified by adding a colon (`:`) and an integer constant after a **member_definition** (see **Structs** lecture):

```
struct flags {
  unsigned int flag1: 1;
  unsigned int flag2: 1;
  ...
  unsigned int flag8: 1;
};
```

This struct theoretically requires just one byte!

(Types have been changed from `int` to `unsigned int`, because a single bit cannot have a sign bit.)

Bit-Fields (contd.)

In reality, more space will generally be required due to alignment and related CPU instruction issues.

Bit-fields are implemented using machine instructions that operate on the bits in units such as *words*, so the minimum width of storage will be at least one word typically.

For example, the above bit-field `flags` struct takes 4 bytes on an x86_64 system using GCC.

Of course the earlier `int` members version of `flags` takes 32 bytes, so there is still a substantial savings.

Bit-Fields (contd.)

Many elements of bit-fields are left to the implementation, so may vary between compilers and between CPU architectures.

In particular, the order bit-field elements are stored is implementation dependent, and can be affected by specifics of the fields, CPU **endianess**, etc.

This means that programers must be very careful when using bit-fields, especially when code must be portable!

And of course it is up to the programmer to ensure values stored in bit-fields are appropriate for the number of bits and type.

E.g., consider a *4-bit* `int` bit-field member; here are how certain assigned values may end up being stored/interpreted:
10 $\Rightarrow$ -6, 15 $\Rightarrow$ -1, 17 $\Rightarrow$ 1, 20 $\Rightarrow$ 4.

Bit-Fields (contd.)

Since bit-fields may not be on byte-boundaries, you are not allowed to get the *address* of a bit-field member of a struct.

Alignment affected by **unnamed, zero-width** bit-field:

- declaration omits identifier, specifies width of 0 (zero)
- indicates that no further bit-field is to be packed into the unit the previous bit-field was stored in
- next bit-field is to use a new storage unit, so be *aligned*

Example:

```
struct flags {
  unsigned int flag1:1;  //aligned, start of bit-field object
  unsigned int flag2:1;
  int :0;
  unsigned int flag3:1;  //aligned
  unsigned int flag4:1;
};
```

Bit-Fields (contd.)

Key points from the C99 standard:

- "An implementation may allocate any *addressable storage unit* large enough to hold a bit-field."
- "The alignment of the addressable storage unit is unspecified."
- "If enough space remains, a bit-field that immediately follows another bit-field in a structure shall be packed into adjacent bits of the same unit. If insufficient space remains, whether a bit-field that does not fit is put into the next unit or overlaps adjacent units is implementation-defined."
- "The order of allocation of bit-fields within a unit (high-order to low-order or low-order to high-order) is implementation-defined."
- "Within a structure object, the non-bit-field members and the units in which bit-fields reside have addresses that increase in the order in which they are declared."
- "A pointer to a structure object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides), and vice versa."