Advanced C Workshop

Lecture 0

Preface

Most of the lectures in this workshop will deal with concepts that's not as inclusive of all architectures. One of such basic assumptions we come across throughout this lecture is the size of various data types. The size of data types in C are different in x86 and x64 architectures; the size of a pointer in x64 is 8 bytes and size of pointer in x86 is 4 bytes. To address every concept for every architecture out there is nigh impossible. Hence, for the sake of time and consistency, most of every lecture in this workshop assumes the architecture to be x64. That's not to say exceptions might not be made. But when done so, it will be mentioned properly.

Back to basics, lifetime

Each non-static stack based variable has a lifetime attached to the scope it's been declared, meaning a variable will "die" after its scope has been passed. Which is why we need to be extra careful when returning pointers to those variables since they might be dangling and we'll be treading into uncharted territory. On the contrary, returning string literals as char pointers is good because their lifetime is attached to the program's runtime instead of any given local scope.

Endianness

Endianness is the order or sequence of bytes stored in digital memory, primarily divided into Big-Endian**(BE)** and Little-endian**(LE)** (and in some very rare cases Middle-Endian). Most of the devices(x86, most AMD processors,etc.) are little-endian while big-endian are dominant in networking protocols.

Big-Endian refers to storing the most significant bytes in lower memory addresses and successive significant bytes in increasing memory addresses i.e. the number (e.g. 0x123456 is stored as 0x12 0x34 0x56). Whereas little endian refers to storing the least significant bytes in lower memory addresses (e.g. 0x123456 is stored as 0x56 0x34 0x12).

As most processors are little endian, unless we're dealing with cross architecture data communication or transfer of data over the internet, endianness isn't a concern for us.

Neither architecture has a huge lead over the other in terms of capability and efficiency.

Arithmetic operations(like addition, subtraction and multiplication) start with the least significant bytes and carry their results into the operations for bytes of increasing significance which is

essentially a little endian operation. Whereas for things like division and comparison we operate on the most significant bytes first and then work down the ladder towards the bytes of lower significance which is a big endian operation.

On modern/large scale processors (i.e. the ones on our computers and laptops), which perform on multiple bytes at once aren't really affected by the endianness. But several microprocessors and microcontrollers operate on small scale byte-addressing, this has a few performance penalties associated with the operations which have a preferred order of treatment. So several non-portable low level optimizations need to be performed pertaining to the byte order of the system.

Bitfields:

A bit field is a data structure that consists of one or more adjacent bits which have been allocated to hold a sequence of bits, stored so that any single bit or group of bits within the group can be set or inspected. A bit field is most commonly used to represent primitive types such as ints, chars, shorts and long longs and hence are of a fixed width.

They are mostly used to reduce memory consumption when a program requires a number of integer variables which always will have low values or in some cases when a bunch of flags need to be stored/interrogated.

As an example, let's say we had to create a logger (used to log information in a managed way). The logger will require several booleans to store what the log level should be, i.e. it can be errors only, or it can be info only or it can be info only and so on. While we could handle this by using a couple of booleans, using bit fields will make this compact.

```
#include <stdio.h>
#include <stdbool.h>

typedef struct { //without bitfields
      bool warn;
      bool error;
      bool info;
} Logger;

typedef enum {  //with bitfields
      NONE = 0,            //0b0000
      LOG_INFO = 1,        //0b0001
      LOG_WARN = 1 << 1,   //0b0010
      LOG_ERROR = 1 << 2,  //0b0100
      LOG_ERROR_AND_WARN = LOG_ERROR | LOG_WARN,  //0b0110
```

```
	LOG_ALL = LOG_INFO | LOG_ERROR | LOG_WARN, //0b0111
} LOGGER;


int main()
{
	Logger logger;
	logger.warn = true;
	logger.error = true;
	logger.info = true;

	LOGGER log = LOG_ALL;

	if(logger.warn && logger.error && logger.info) {
		//do something
	} else if (logger.warn && logger.error) {
		//do something
	} else if (logger.error) {
		//do something
	} else if (logger.info) {
		//do something
	} else if (logger.warn) {
		//do something
	}


	switch (log) {
		case LOG_ALL: break;
		case LOG_ERROR_AND_WARN: break;
		case LOG_ERROR: break;
		case LOG_WARN: break;
		case LOG_INFO: break;
		case NONE: break;
	}
}
```

Padding

Memory plays a big part in programs, and is usually the culprit in bottlenecking your system resources, looking at you chrome 🔪. And padding is one of the reasons why this happens.

Padding occurs when the order of declaration of variables messes up the alignment of the memory occupied by said variables. It depends on the architecture of the cpu (i.e. x86 vs x64) as well as the type of the variables declared.

Any data type in C (except char) has a specific set of requirements of alignment, chars can start on any byte address, but 2-byte shorts must start on an even address, 4-byte ints or floats must start on an address divisible by 4, and 8-byte longs or doubles must start on an address divisible by 8. This requirement is called self-alignment. Padding plays a huge part when dealing with structures and arrays of structures.

Padding is the result of optimization in accessing variables. If the memory is aligned, then the access will be fast.
Characters are a special case; they're equally expensive from anywhere they live inside a single machine word. Which is why they don't have a preferred alignment.

Now we'll look at a simple example of variable layout in memory. Consider the following series of variable declarations in the top level of a C module:

```
char *p;
char c;
int x;
```

If we didn't know anything about data alignment, we might assume that these three variables would occupy a continuous span of bytes in memory. 8 bytes of pointer would be immediately followed by 1 byte of char and that immediately followed by 4 bytes of int.

Here's what actually happens. The storage for p starts on a self-aligned 8-byte boundary. This is pointer alignment - the strictest possible. The storage for c follows immediately. But the 8-byte alignment requirement of x forces a gap in the layout; it comes out as though there were a fourth intervening variable, like this:

```
char *p;          /* 8 bytes */
char c;           /* 1 byte */
char pad[3];      /* 3 bytes */
int x;            /* 4 bytes */
```

The pad[3] character array represents the fact that there are three bytes of waste space in the structure. The old-school term for this was "slop". The value of the padding bits is undefined; in particular it is not guaranteed that they will be zeroed.

The padding is of fixed size here as it starts on an 8-byte boundary (as the first variable is a pointer), but that's not the case when the type with strictest alignment requirements isn't in the top.

Consider this

```
char c;
char *p;
int x;
```

If the actual memory layout were written like this

```
char c;
char pad1[M];
char *p;
char pad2[N];
int x;
```

What can we say about M and N?

First, in this case N will be zero. The address of x, coming right after p, is guaranteed to be pointer-aligned, which is never less strict than int-aligned.

The value of M is less predictable. If the compiler happened to map c to the last byte of a machine word, the next byte (the first of p) would be the first byte of the next one and properly pointer-aligned. M would be zero.

It is more likely that c will be mapped to the first byte of a machine word. In that case M will be 7 to ensure that p has a pointer alignment 64-bit machine.

Intermediate cases are possible. M can be anything from 0 to 7 (0 to 3 on 32-bit) because a char can start on any byte boundary in a machine word.

If you wanted to make those variables take up less space, you could get that effect by swapping x with c in the original sequence.

```
char *p;    /* 8 bytes */
int x;      /* 4 bytes */
char c;     /* 1 byte */
```

Usually, for the small number of scalar variables in your C programs, bumming out the few bytes you can get by changing the order of declaration won't save you enough to be significant. The technique becomes more interesting when applied to nonscalar variables - especially structs.

Before we get to those, let's dispose of arrays of scalars. On a platform with self-aligned types, arrays of char/short/int/long/pointer have no internal padding; each member is automatically self-aligned at the end of the next one.

Structure alignment and padding

In general, a struct instance will have the alignment of its widest scalar member. Compilers do this as the easiest way to ensure that all the members are self-aligned for fast access.
But on top of the normal padding between the member variables, trail padding is something we need to consider because of arrays.

Consider this struct:

```
struct Foo {
    char *p;
    long x;
    char c;
};
```

Because the largest scalar in the struct is a pointer, and since the starting address of the struct instance is the same as the starting address of the first member, the struct will be aligned to the address the pointer is aligned to. The size of the struct is 16 bytes. This is because arrays are a thing and each member of an array too needs to be self aligned, and hence there's 3 bytes of padding.
The above structure is stored in memory as follows:

```
struct Foo {
    char *p;      //8 bytes
    long x;       //4 bytes
    char c;       //1 byte
    char pad[3];  //3 bytes -trailing padding
};
```

And lastly, this struct:

```
struct Bar {
    char c;
    struct Baz {
        char *p;
        short x;
    } baz;
};
```

Here the struct Baz needs to be aligned to the widest scalar i.e. the pointer, and since Bar contains an element that is aligned to a multiple of 8 bytes, it too needs to be aligned to a memory address that is a multiple of 8 bytes. So the char c, needs to be aligned to a 8 byte aligned memory address as well. As a whole the struct bar takes up 24 bytes of which 13 bytes are padding which brings us to our next topic structure re-ordering.
The above structure is stored in memory as follows:

```
struct Bar {
    char c;           //1 byte
    char pad1[7];     //7 bytes
    struct Baz {
        char *p;      //8 bytes
        short x;      //2 bytes
        char pad2[6]; //6 bytes
    } baz;
};
```

Structure Reordering

To save the memory space waste due to padding, we employ certain reordering techniques of structure members. This is often referred to as structure packing.

The simplest way to achieve this is to reorder the structure members by decreasing alignment. That is: make all the pointer-aligned subfields come first, because on a 64-bit machine they will be 8 bytes. Then the 4-byte ints; then the 2-byte shorts; then the character fields.

Consider this simple linked-list structure:

```
struct Foo {
      char c;
      struct Foo *p;
      short x;
};
```

This is laid on memory as follows:

```
struct Foo {
      char c;           /* 1 byte */
      char pad1[7];     /* 7 bytes */
      struct Foo* p;    /* 8 bytes */
      short x;          /* 2 bytes */
```

```
      char pad2[6];     /* 6 bytes */
};
```

That's 24 bytes. If we reorder by size, we get this:

```
struct Bar {
    struct Bar *p;      /* 8 bytes */
    short x;            /* 2 bytes */
    char c;             /* 1 byte  */
};
```

Considering self-alignment, we see that none of the data fields need padding. This is because the stride address for a (longer) field with stricter alignment is always a validly-aligned start address for a (shorter) field with less strict requirements. All the repacked struct actually requires is trailing padding. And finally the repack transformation drops the size from 24 to 16 bytes.

```
struct Bar {
      struct Bar* p;    /* 8 bytes */
      short x;          /* 2 bytes */
      char c;           /* 1 byte */
      char pad[5];      /* 5 bytes */
};
```

> Note: Reordering is not guaranteed to produce savings.

C is a language originally designed for writing operating systems and other code close to the hardware. Automatic reordering often interferes with a systems programmer's ability to lay out structures that exactly match the byte and bit-level layout of memory-mapped device control blocks. For example, when creating files of set formats (e.g. bitmaps and pngs) the metadata needs to be in a certain order which has to be set according to the standard and not willy-nilly to decrease the struct size. Hence, such reordering is not done by the compiler even if it seems like a trivial task.

Structure bitfields

To expand on bitfields, we consider following structure

```
struct Foo {
      short s;
      char c;
      int flip:1;
```

```
      int nibble:4;
      int septet:7;
};
```

Bitfields are implemented with word and byte-level masks and rotate instructions operating on machine words, and cannot cross word boundaries.

Assuming we're on a 32-bit machine, the C99 rules imply that the layout may look like this:

```
struct Foo {
      short s;    /* 2 bytes */
      char c;     /* 1 byte */
      int flip:1; /* total 1 bit */
      int nibble:4;  /* total 5 bits */
      int pad1:3; /* pad to an 8-bit boundary */
      int septet:7;  /* 7 bits */
      int pad2:25;   /* pad to 32 bits */
};
```

But this isn't the only possibility, because the C standard does not specify that bits are allocated low-to-high. So the layout could look like this:

```
struct Foo {
      short s;    /* 2 bytes */
      char c;     /* 1 byte */
      int pad1:3; /* pad to an 8-bit boundary */
      int flip:1; /* total 1 bit */
      int nibble:4;  /* total 5 bits */
      int pad2:25;   /* pad to 32 bits */
      int septet:7;  /* 7 bits */
};
```

That is, the padding could precede rather than following the payload bits.

Note also that, as with normal structure padding, the padding bits are not guaranteed to be zero.

Readability and cache locality

While reordering by size is the simplest way to eliminate padding, it's not necessarily the right thing. There are two more issues: readability and cache locality.

Programs are not just communications to a computer, they are communications to other human beings. Code readability is important. A clumsy, mechanical reordering of your structure can harm readability. When possible, it is better to reorder fields so they remain in coherent groups with semantically related pieces of data kept close together.

When the program frequently accesses a structure, or parts of a structure, it is helpful for performance if the accesses tend to fit within a cache line (the memory block fetched by your processor when it is told to get any single address within the block). On 64-bit x86 a cache line is 64 bytes beginning on a self-aligned address; on other platforms it is often 32 bytes.

Doing things like grouping related and co-accessed data in adjacent fields to improve readability also improves cache-line locality. These are both reasons to reorder intelligently, with awareness of your code's data-access patterns.

Overriding alignment rules

Sometimes we need a specific order in which the bytes need to be set (e.g. the metadata of any file format), which might be ruined by the padding present for fast access. We can take the performance hit and override the alignment rules by using the #pragma pack(n) directive. Where n = 1,2,4,8,16 is the option that allows us to change the alignment of data types within a struct to align to boundaries smaller than its size.

C operators alignof and alignas

We can set our own alignment for any variable and check the existing alignment using _Alignas (alignas is the macro form) and _Alignof(alignof is the macro form) operators.

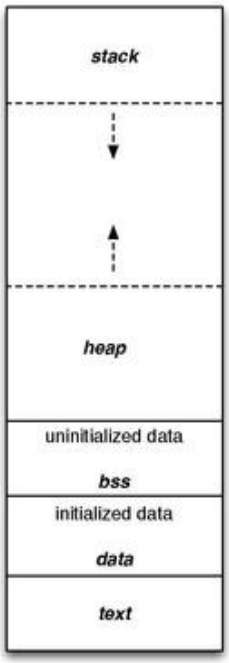
```
#include <stdalign.h>
#include <stdio.h>

// every object of type struct sse_t will be aligned to 16-byte boundary
struct sse_t {
  alignas(16) float sse_data[4];
};

// every object of type struct data will be aligned to 128-byte boundary
struct data {
  char x;
  alignas(128) char cacheline[128]; // over-aligned array of char not array
of over-aligned chars
};
```

```c
int main(void) {
    printf("sizeof(data) = %zu (1 byte + 127 bytes padding + 128-byte
array)\n", sizeof(struct data));
     printf("alignment of sse_t is %zu\n", alignof(struct sse_t));
}
```

Memory Segments of a program

Most object file formats are structured as separate sections of data, each section containing a certain type of data. When a program is loaded into memory by a loader, the loader allocates various regions of memory to the program. Some of these regions correspond to segments of the object file, and thus are usually known by the same names. Others, such as the stack, only exist at run time. In some cases, relocation is done by the loader (or linker) to specify the actual memory addresses.

A typical memory representation of a C program consists of the following sections.
1. Text segment (i.e. instructions)
2. Initialized data segment
3. Uninitialized data segment (bss)
4. Heap
5. Stack



Code Segment

The code segment (aka the text segment), is a portion of an object file or the corresponding section of the program's address space that contains executable instructions.

The code segment in memory is typically read-only and has a fixed size, so on embedded systems it can usually be placed in the ROM, without the need for loading. As a memory region, the code segment may be placed below the heap or stack in order to prevent heap and stack overflows from overwriting it.

Data Segment

The data segment (often denoted .data) is a portion of an object file or the corresponding address space of a program that contains initialized static variables, that is, global variables and static local variables. The size of this segment is determined by the size of the values in the program's source code, and does not change at run time.

The data segment is read/write, since the values of variables can be altered at run time. This is in contrast to the read-only data segment (rodata segment or .rodata), which contains static constants rather than variables; it also contrasts to the code segment, which is read-only on many architectures. Uninitialized data, both variables and constants, is instead in the BSS segment.

BSS Segment

The block starting symbol (abbreviated to .bss or bss) is the portion of an object file or executable that contains statically allocated variables that are declared but have not been assigned a value yet. It is often referred to as the "bss section" or "bss segment".

Typically only the length of the bss section, but no data, is stored in the object file. The program loader allocates memory for the bss section when it loads the program. By placing variables with no value in the .bss section, instead of the .data or .rodata section which require initial value data, the size of the object file is reduced.

Unix-like systems and Windows initialize the bss section to zero, allowing C and C++ statically allocated variables initialized to values represented with all bits zero to be put in the bss segment.

Stack

The stack segment contains the call stack, a LIFO(Last In, First Out) structure that stores information about the active subroutines of a computer program, typically located in the higher parts of memory. A "stack pointer" register tracks the top of the stack; it is adjusted each time a

value is "pushed" onto the stack. The set of values pushed for one function call is termed a "stack frame". A stack frame consists at minimum of a return address. Automatic variables are also allocated on the stack.

The stack segment traditionally adjoins the heap segment and they grow towards each other; when the stack pointer meets the heap pointer, free memory is exhausted. With large address spaces and virtual memory techniques they tend to be placed more freely, but they still typically grow in a converging direction.

The stack size by default is 4 MB but can be changed by setting the appropriate flags.

Heap

The heap is a free store of memory which provides ways to dynamically allocate portions of memory to programs at the programmer's request, and free it for reuse when no longer needed. Unlike stack allocated memory, heap allocated memory's lifetime is not tied to the scope. It lies below the stack(i.e it begins at the end of the BSS segment) and grows upwards towards the stack, and is usually the region where huge chunks of data are stored (e.g. the pixel data of an image or the vertices of a 3d model). Though it is to be kept in mind that access to memory in the heap is slower than the access to the memory in the stack.

Malloc, realloc and free

Heap allocation is managed by the standard lib functions: malloc, calloc, realloc, and free, which in turn call into various OS specific syscalls (VirtualAlloc in windows and mmap in linux).

Malloc

It allocates size bytes of uninitialized storage in heap. If allocation succeeds, it returns a pointer that is suitably aligned for any object type with fundamental alignment.
If size is zero, the behavior of malloc is implementation-defined. For example, a null pointer may be returned. Alternatively, a non-null pointer may be returned; but such a pointer should not be dereferenced, and should be passed to free to avoid memory leaks.

Realloc

It takes in a pointer to a memory previously allocated and copies the memory present there into a new memory location, calls free on the previous location and returns the pointer to the new location. It basically resizes the memory. And in case a null pointer is passed, it acts just like malloc.

Free

It is used to free the memory previously allocated on the heap.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
	int *p1 = malloc(4 * sizeof(int));  // allocates enough for an array of 4 ints
	int *p2 = malloc(4 * sizeof *p2);   //or we can simply do this without specifying the type and just using the variable name.

	if(p1) {
	for(int n = 0; n < 4; ++n) // populate the array
		p1[n] = n * n;
	for(int n = 0; n < 4; ++n) // print it back out
		printf("p1[%d] == %d\n", n, p1[n]);
	}

	p2 = realloc(p2, 10 * sizeof(*p2)); //incase the initially allocated memory wasn't enough
	if(p2) {
	for(int n = 0; n < 10; ++n)
		p1[n] = n * n;
	for(int n = 0; n < 10; ++n)
		printf("p2[%d] == %d\n", n, p1[n]);
	}

	if(p1) free(p1);
	if(p2) free(p2);
}
```

Pointer alignment

Similar to variables being self aligned for fast memory access, pointers too need to be self aligned for efficient access. Any pointer returned by a memory allocation function is self aligned and you don't have to care for alignment, but the case might not be true for when you get into intricate casts from a less strict type to a more strict type. Consider the example,

```
char c;
int* ptr = &c;
```

Here, if the char c wasn't aligned to a 4 byte aligned address, casting to a stricter type will cause for a non aligned access, hence casting a char to an integer and then dereferencing that pointer isn't something that's recommended.This can lead to various issues from slower access speed to program crashes(on specific hardwares).

References

- Bitwise Operators Part 1
- Bitwise Operators Part 2
- Bitfields
- Alignment and Padding
- Object (extra reading)
- _Alignas operator
- Details on Alignment
- Memory layout
- Code segment
- Data segment
- .bss
- Stack pointer (MIPS Architecture) - optional, to be covered in lecture 2