Alignment in C

Seminar “Effiziente Programmierung in C”

Sven-Hendrik Haase

Universität Hamburg, Fakultät für Informatik

2014-01-09

Outline

Introduction

Guiding Questions of This Presentation

- Which types of alignment exist in C?

Introduction

Guiding Questions of This Presentation

- Which types of alignment exist in C?
- What is data alignment?

Introduction

Guiding Questions of This Presentation

- Which types of alignment exist in C?
- What is data alignment?
- What is heap alignment?

Introduction

Guiding Questions of This Presentation

- Which types of alignment exist in C?
- What is data alignment?
- What is heap alignment?
- What is stack alignment?

Introduction

Guiding Questions of This Presentation

- Which types of alignment exist in C?
- What is data alignment?
- What is heap alignment?
- What is stack alignment?
- How does it work in C?

Introduction

Guiding Questions of This Presentation

- Which types of alignment exist in C?
- What is data alignment?
- What is heap alignment?
- What is stack alignment?
- How does it work in C?
- Do we need to care abouy any of these?

Introduction

Memory Addressing

- Computers address memory in word-sized chunks

Introduction

Memory Addressing

- Computers address memory in word-sized chunks
- A **word** is a computer's natural unit for data
- Word size is defined by architecture
- Usual word sizes: 4 bytes on 32-bit, 8 bytes on 64-bit

Introduction

Memory Addressing

- Computers address memory in word-sized chunks
- A **word** is a computer's natural unit for data
- Word size is defined by architecture
- Usual word sizes: 4 bytes on 32-bit, 8 bytes on 64-bit
- This means we can only address data at memory locations that are multiples of 4 or 8 respectively (strictly speaking)
- Many processors allow access of arbitrary memory locations while some fail horribly

Introduction

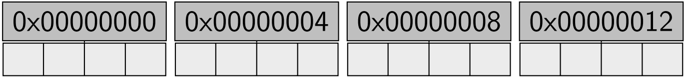
Memory Addressing

- Modern processors can load word-sized (4 bytes) and long word-sized (8 bytes) memory locations equally well
- Find out word-sizes:
  - `getconf WORD_BIT` (32 for me, 32 on RPi)
  - `getconf LONG_BIT` (64 for me, 32 on RPi)

Introduction

Alignment 101

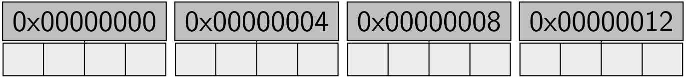
- Assume a 32-bit architecture with a word size of 4 byte



Introduction

Alignment 101

- Assume a 32-bit architecture with a word size of 4 byte

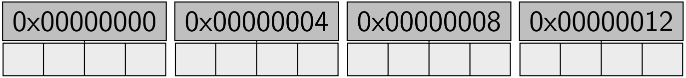


- Let's save a 4 byte **int**  in our memory:

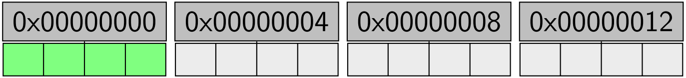
Introduction

Alignment 101

- Assume a 32-bit architecture with a word size of 4 byte



- Let's save a 4 byte **int**  in our memory:



- Looks good!

Introduction

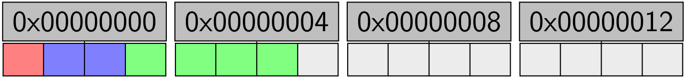
Alignment 101

- Let's save a **char** , a **short**  and an **int**  in our memory:

Introduction

Alignment 101

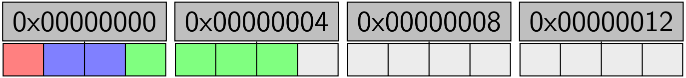
- Let's save a **char**, a **short** and an **int** in our memory:



Introduction

Alignment 101

- Let's save a **char**, a **short** and an **int** in our memory:

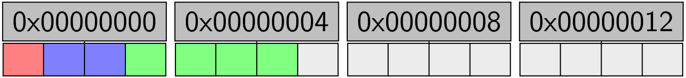


- Oh wait

Introduction

Alignment 101

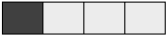
- Let's save a **char**, a **short** and an **int** in our memory:



- Oh wait
- Needs two memory accesses and some arithmetic to fetch the **int**.

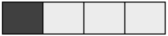
Introduction

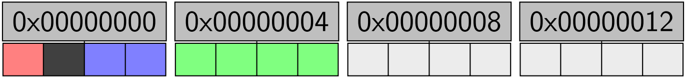
Alignment 101

- We need to be smarter about this!
- Padding  to the rescue

Introduction

Alignment 101

- We need to be smarter about this!
- Padding  to the rescue



- Much better
- This is considered **naturally aligned**

Introduction

Consquences of Misalignment

- Different behavior depending on architecture
- Alignment fault errors on some platforms (RISC, ARM)
- Bad performance on others
- SSE requires proper alignment per specification (though this restriction is about to be removed)

Introduction

Different Types of Alignment

- Some definitions so we don't get confused:

Introduction

Different Types of Alignment

- Some definitions so we don't get confused:
- **Data Structure Alignment** refers to the alignment of sequential memory inside a data structure (struct)

Introduction

Different Types of Alignment

- Some definitions so we don't get confused:
- **Data Structure Alignment** refers to the alignment of sequential memory inside a data structure (struct)
- **Heap Alignment** refers to the alignment of dynamically allocated memory

Introduction

Different Types of Alignment

- Some definitions so we don't get confused:
- **Data Structure Alignment** refers to the alignment of sequential memory inside a data structure (struct)
- **Heap Alignment** refers to the alignment of dynamically allocated memory
- **Stack Alignment** refers to the alignment of the stack pointer

Data Structure Alignment

Structs and Stuff

Data Structure Alignment

Structs and Stuff

Consider this:

```
struct Foo {
    char x; // 1 byte
    short y // 2 bytes
    int z; // 4 bytes
};
```

- The struct's naive size would be 1 byte + 2 bytes + 4 bytes = 7 bytes

Data Structure Alignment

Structs and Stuff

Consider this:

```
struct Foo {
    char x; // 1 byte
    short y // 2 bytes
    int z; // 4 bytes
};
```

- The struct's naive size would be 1 byte + 2 bytes + 4 bytes = 7 bytes
- Of course, we know it's actually going to be

Data Structure Alignment

Structs and Stuff

Consider this:

```
struct Foo {
    char x; // 1 byte
    short y // 2 bytes
    int z; // 4 bytes
};
```

- The struct's naive size would be 1 byte + 2 bytes + 4 bytes = 7 bytes
- Of course, we know it's actually going to be 8 bytes due to padding

Data Structure Alignment

Structs and Stuff

- **A struct is aligned to the largest type's alignment requirements**

Data Structure Alignment

Structs and Stuff

- **A struct is aligned to the largest type's alignment requirements**
- This can yield some rather inefficient structures:

```
struct Foo {
    char x; // 1 byte
    double y // 8 bytes
    char z; // 1 bytes
};
```

- The struct's naive size would be 1 byte + 8 bytes + 1 bytes = 10 bytes

Data Structure Alignment

Structs and Stuff

- **A struct is aligned to the largest type's alignment requirements**
- This can yield some rather inefficient structures:

```
struct Foo {
    char x; // 1 byte
    double y // 8 bytes
    char z; // 1 bytes
};
```

- The struct's naive size would be 1 byte + 8 bytes + 1 bytes = 10 bytes
- Its effective size is

Data Structure Alignment

Structs and Stuff

- **A struct is aligned to the largest type's alignment requirements**
- This can yield some rather inefficient structures:

```
struct Foo {
    char x; // 1 byte
    double y // 8 bytes
    char z; // 1 bytes
};
```

- The struct's naive size would be 1 byte + 8 bytes + 1 bytes = 10 bytes
- Its effective size is 24 bytes!

Data Structure Alignment

Structs and Stuff

- The memory ineffiency can be minimized by reordering the members like so:

```
struct Foo {
    char x; // 1 byte
    char z; // 1 bytes
    double y // 8 bytes
};
```

- Now it's only 16 bytes, best we can do if we want to keep alignment

Data Structure Alignment

Structs and Stuff

- How about this?

```
struct Foo {
    double a; // 8 byte
    char b; // 1 byte
    char c; // 1 byte
    short d; // 2 bytes
    int e; // 4 bytes
    double f; // 8 bytes
};
```

Data Structure Alignment

Structs and Stuff

- How about this?

```
struct Foo {
    double a; // 8 byte
    char b; // 1 byte
    char c; // 1 byte
    short d; // 2 bytes
    int e; // 4 bytes
    double f; // 8 bytes
};
```

- This structure is 24 bytes in total
- Most efficient configuration possible
- It's called **tighly packed**

Data Structure Alignment

Structs and Stuff

- How about extension types?

```
struct Foo {
    char x; // 1 byte
    __uint128_t y; // 16 bytes
    char a; // 1 byte
    __uint128_t b; // 16 bytes
};
```

- This struct is

Data Structure Alignment

Structs and Stuff

- How about extension types?

```
struct Foo {
    char x; // 1 byte
    __uint128_t y; // 16 bytes
    char a; // 1 byte
    __uint128_t b; // 16 bytes
};
```

- This struct is 64 bytes
- World's most wasteful struct

Data Structure Alignment

Structs and Stuff

- Of course, we can also reorder this to make it 34 bytes only

```
struct Foo {
    __uint128_t y; // 16 bytes
    __uint128_t b; // 16 bytes
    char x; // 1 byte
    char a; // 1 byte
};
```

Data Structure Alignment

Padding in the Real World

- Every decent compiler will automatically use data structure padding depending on architecture

Data Structure Alignment

Padding in the Real World

- Every decent compiler will automatically use data structure padding depending on architecture
- Some compilers support `-Wpadded` which generates nice warnings about structure padding

Data Structure Alignment

Padding in the Real World

- Every decent compiler will automatically use data structure padding depending on architecture
- Some compilers support -Wpadded which generates nice warnings about structure padding
- Compiler warnings can help you find inefficiencies
- Example output with clang:

```
clang -Wpadded -o example1 example1.c
example1.c:5:11: warning: padding struct
'struct Foo' with 1 byte to align 'y' [-Wpadded]
short y;
      ^
1 warning generated.
```

Data Structure Alignment

Padding in the Real World

- It's possible to prevent the compiler from padding a struct using either `__attribute__((packed))` after a struct definition, `#pragma pack (1)` in front of a struct definition or `-fpack-struct` as a compiler parameter

Data Structure Alignment

Padding in the Real World

- It's possible to prevent the compiler from padding a struct using either `__attribute__((packed))` after a struct definition, `#pragma pack (1)` in front of a struct definition or `-fpack-struct` as a compiler parameter
- Either of these generate an incompatible ABI
- We can use the `sizeof` operator to check the effective size of a struct

Data Structure Alignment

Performance Implications

- Do we actually have to worry about this?

Data Structure Alignment

Performance Implications

- Do we actually have to worry about this?
- Most likely not unless in special use cases (device drivers, extremely memory limited computers) or when using a compiler from 1878

Data Structure Alignment

Performance Implications

For fun, let's look at the performance impact of misaligned memory:

```
struct Foo {
    char x;
    short y;
    int z;
};

struct Foo foo;
clock_gettime(CLOCK, &start);
for (unsigned long i = 0; i < RUNS; ++i) {
     foo.z = 1;
     foo.z += 1;
}
clock_gettime(CLOCK, &end);
```

```
struct Bar {
    char x;
    short y;
    int z;
} __attribute__((packed));

struct Bar bar;
clock_gettime(CLOCK, &start);
for (unsigned long i = 0; i < RUNS; ++i) {
     bar.z = 1;
     bar.z += 1;
}
clock_gettime(CLOCK, &end);
```

Compiled with

```
gcc -DRUNS=400000000 -DCLOCK=CLOCK_MONOTONIC -std=gnu99 -O0
```

Data Structure Alignment

Performance Implications

Results

aligned runtime: 9.504220399 s
unaligned runtime: 9.491816620 s

Data Structure Alignment

Performance Implications

Results

aligned runtime: 9.504220399 s
unaligned runtime: 9.491816620 s

- Takes the same time!

Data Structure Alignment

Performance Implications

Results

aligned runtime: 9.504220399 s
unaligned runtime: 9.491816620 s

- Takes the same time!
- Nowadays it totally doesn't matter for performance! :D
- Modern processors can read aligned/unaligned memory equally fast (at least Intel Sandy Bridge and up)

Data Structure Alignment

Performance Implications

Results

aligned runtime: 9.504220399 s
unaligned runtime: 9.491816620 s

- Takes the same time!
- Nowadays it totally doesn't matter for performance! :D
- Modern processors can read aligned/unaligned memory equally fast (at least Intel Sandy Bridge and up)
- But what about processors with the computing power of a potato?

Data Structure Alignment

Performance Implications

Results on Raspberry Pi with 1/10 the loop length

aligned runtime: 12.174631568 s
unaligned runtime: 26.453561832 s

Data Structure Alignment

Performance Implications

Results on Raspberry Pi with 1/10 the loop length

aligned runtime: 12.174631568 s
unaligned runtime: 26.453561832 s

- On some architectures alignment matters a lot!
- We can nicely see that it takes about twice the time (two memory fetches) + some arithmetic

Data Structure Alignment

SSE

- Classically, SSE requires 16 byte alignment of data and stack pointer
- Requirement will be lifted soon

Data Structure Alignment

SSE

- Classically, SSE requires 16 byte alignment of data and stack pointer
- Requirement will be lifted soon
- Compilers automatically align to that when using SIMD types (`__m128` and friends)
- x86_64 is 16 byte aligned anyway
- Very modern compilers even automagically vectorize loops
- No worries to the programmer ☺

Heap Alignment

Introduction

Heap Alignment

Introduction

- `malloc` is usually good enough
- Allocated memory is aligned to largest primitive type

Heap Alignment
Introduction

- `malloc` is usually good enough
- Allocated memory is aligned to largest primitive type
- Use `aligned_alloc` instead of `malloc` for custom alignments
- Other heap alignment functions: `posix_memalign`, `aligned_alloc` and `valloc`

Heap Alignment

Introduction

- malloc is usually good enough
- Allocated memory is aligned to largest primitive type
- Use aligned_alloc instead of malloc for custom alignments
- Other heap alignment functions: posix_memalign, aligned_alloc and valloc
- memalign and pvalloc are considered obsolete

Heap Alignment

Example

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

#define SIZE 1024 * 1024
#define ALIGN 4096
int main()
{
    void* a = malloc(SIZE);
    void* b = aligned_alloc(ALIGN, SIZE);

    printf("a: %p, a %% %i: %lu\n", a, ALIGN, ((unsigned long)a) % ALIGN);
    printf("b: %p, b %% %i: %lu\n", b, ALIGN, ((unsigned long)b) % ALIGN);
    return 0;
}
```

Results

a: 0x7fdec2265010, a % 4096: 16

b: 0x7fdec1cec000, b % 4096: 0

Heap Alignment

Use Cases

You should consider using custom heap memory alignments when. . .

Heap Alignment

Use Cases

You should consider using custom heap memory alignments when. . .

- interfacing with low-level stuff (hardware)

Heap Alignment

Use Cases

You should consider using custom heap memory alignments when. . .

- interfacing with low-level stuff (hardware)
- trying to be really clever about CPU cache line optimization

Heap Alignment

Use Cases

You should consider using custom heap memory alignments when...

- interfacing with low-level stuff (hardware)
- trying to be really clever about CPU cache line optimization
- writing custom allocators (for instance when writing an interpreter or garbage collector)

Heap Alignment

Use Cases

You should consider using custom heap memory alignments when...

- interfacing with low-level stuff (hardware)
- trying to be really clever about CPU cache line optimization
- writing custom allocators (for instance when writing an interpreter or garbage collector)
- using SIMD and your compilers is too stupid to align stuff properly by itself

Stack Alignment

Introduction

Stack Alignment

Introduction

- Different platforms make different assumptions about stack alignment
- Platforms:
  - Linux: depends (legacy is 4 byte, modern is 16 byte)
  - Windows: 4 byte
  - OSX: 16 byte
  - x86_64 always uses 16 byte

Stack Alignment

Introduction

- Different platforms make different assumptions about stack alignment
- Platforms:
  - Linux: depends (legacy is 4 byte, modern is 16 byte)
  - Windows: 4 byte
  - OSX: 16 byte
  - x86_64 always uses 16 byte
- But why do we care?

Stack Alignment

Introduction

- Different platforms make different assumptions about stack alignment
- Platforms:
  - Linux: depends (legacy is 4 byte, modern is 16 byte)
  - Windows: 4 byte
  - OSX: 16 byte
  - x86_64 always uses 16 byte
- But why do we care?
- Mixing stack alignments is very bad!

Stack Alignment

The Problem

Consider this:

```
void foo() {
    struct MyType bar;
}
```

- Looks benign!

Stack Alignment

The Problem

Consider this:

```
void foo() {
    struct MyType bar;
}
```

- Looks benign!
- Imagine it is 16 byte aligned, then what will happen if this is called from a platform with 4 byte alignment such as Windows?

Stack Alignment

The Problem

Consider this:

```
void foo() {
    struct MyType bar;
}
```

- Looks benign!
- Imagine it is 16 byte aligned, then what will happen if this is called from a platform with 4 byte alignment such as Windows?
- **Stack corruption**

Stack Alignment

The Problem

- We don't usually care about stack alignment unless we have to

Stack Alignment

The Problem

- We don't usually care about stack alignment unless we have to
- If we have cross-architecture calls, we need special tricks

Stack Alignment

The Problem

- We don't usually care about stack alignment unless we have to
- If we have cross-architecture calls, we need special tricks
- To fix, decorate function with `__attribute__((force_align_arg_pointer))` or use `-mstackrealign`

Stack Alignment

The Problem

- We don't usually care about stack alignment unless we have to
- If we have cross-architecture calls, we need special tricks
- To fix, decorate function with `__attribute__((force_align_arg_pointer))` or use `-mstackrealign` (or stop using Windows)
- Other compiler arguments to play with stack alignment: `-mpreferred-stack-boundary`, `-mincoming-stack-boundary`

Stack Alignment

Use Cases

- Play with stack alignment only if you absolutely, positively have to

Stack Alignment

Use Cases

- Play with stack alignment only if you absolutely, positively have to
- Software that needs stack alignment: valgrind (virtual CPU), wine (cross-compiled cross-platform cross-architecture compatibility layer), cross-compilers, kernels

Stack Alignment

Use Cases

- Play with stack alignment only if you absolutely, positively have to
- Software that needs stack alignment: valgrind (virtual CPU), wine (cross-compiled cross-platform cross-architecture compatibility layer), cross-compilers, kernels
- Very memory limited device

Stack Alignment

Use Cases

- Play with stack alignment only if you absolutely, positively have to
- Software that needs stack alignment: valgrind (virtual CPU), wine (cross-compiled cross-platform cross-architecture compatibility layer), cross-compilers, kernels
- Very memory limited device
- You will probably never have to worry about this

Summary

TL;DR

Do worry about

- Positions of members within a struct
- Using weird compiler parameters
- GCC, Windows and SSE instructions

Summary
TL;DR

Do worry about

- Positions of members within a struct
- Using weird compiler parameters
- GCC, Windows and SSE instructions

Do not worry about

- Struct alignment/padding (compilers are smart)
- Performance issues (computers are fast)
- The Stack (unless you are doing really wierd stuff)

Summary

Resources

- http://www.agner.org/optimize/blog/read.php?i=142&v=t
- http://en.wikipedia.org/wiki/Data_structure_alignment
- http://en.wikipedia.org/wiki/Word_(data_type)
- http://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/
- http://lemire.me/blog/archives/2012/05/31/data-alignment-for-speed-myth-or-reality/
- http://www.makelinux.com/books/lkd2/ch19lev1sec3
- http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/aligned.html
- http://tuxsudh.blogspot.de/2005/05/structure-packing-in-gcc.html
- http://www.peterstock.co.uk/games/mingw_sse/
- http://eigen.tuxfamily.org/dox-2.0/WrongStackAlignment.html