Week 9 Lecture 2

NWEN 241
Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

Low-level Systems Programming

Lecture 1:

- Conditional inclusion/compilation
- Standard integer types
- Bit-wise operators

Lecture 2:

- Bit-fields
- Memory alignment
- Structure padding and packing

Accessing bits or groups of bits

Two Approaches:

Traditional C:
Use #define macro in tandem with bitwise operators

Modern:
Use bit-fields

Traditional approach: selecting bits

Suppose

```
uint8_t a = 37;
```

7	6	5	4	3	2	1	0
0	0	1	0	0	1	0	1

How to select / read the lowest 3 bits (bits 0, 1 and 2)?

	0	0	1	0	0	1	0	1
&	0	0	0	0	0	1	1	1
	0	0	0	0	0	1	0	1

Bit mask: indicates bits to be selected

Traditional approach: selecting bits

	0	0	1	0	0	1	0	1
&	0	0	0	0	0	1	1	1
	0	0	0	0	0	1	0	1

Bit mask: indicates bits to be selected

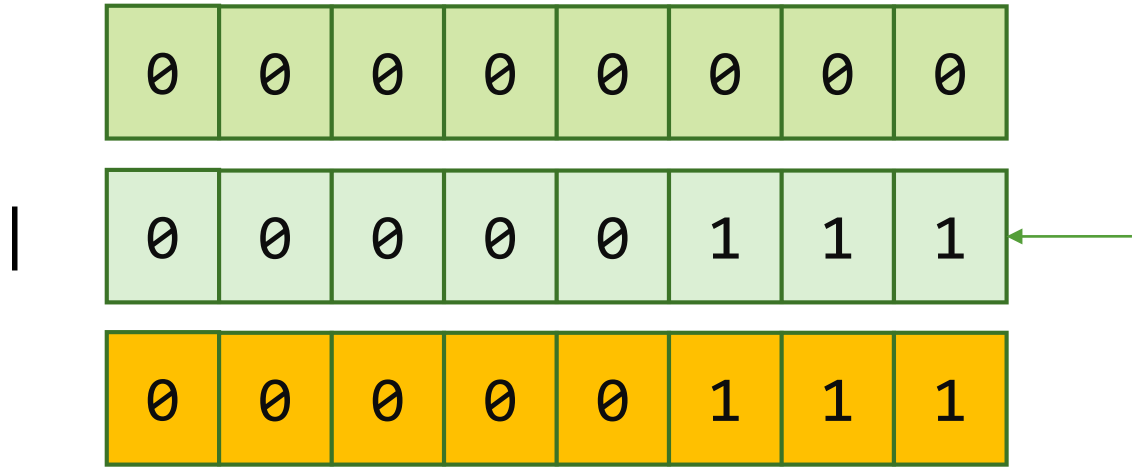
```
uint8_t a = 37;
uint8_t mask = 0x07; // binary 00000111
uint8_t b = a & mask;
```

Binary to hex

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Hex
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Traditional approach: setting bits




```
uint8_t a = 0;
uint8_t mask = 0x07; // binary 00000111
uint8_t b = a | mask;
```

Traditional approach: clearing bits

	1	1	1	1	1	1	1	1	
&	1	1	1	1	1	0	0	0	← Bit mask complement
	1	1	1	1	1	0	0	0	

```
uint8_t a = 255;
uint8_t mask = 0x07; // binary 00000111
uint8_t b = a & ~mask;
```

Traditional approach example

- Serial port line register is an 8-bit read-only register:

7	6	5	4	3	2	1	0

Bit 7: FIFO Error
Bit 6: Empty Data Holding Registers
Bit 5: Empty Transmitter Holding Registers
Bit 4: Break Interrupt
Bit 3: Framing Error
Bit 2: Parity Error
Bit 1: Overrun Error
Bit 0: Data Ready

- When bit is set (1), the condition exists

Traditional approach example

```
#define FIFO_ERROR        0x80 // bit 7 mask
#define EMPTY_DHR         0x40 // bit 6 mask
#define EMPTY_THR         0x20 // bit 5 mask
#define BREAK_INTR        0x10 // bit 4 mask
#define FRAMING_ERROR     0x08 // bit 3 mask
#define PARITY_ERROR      0x04 // bit 2 mask
#define OVERRUN_ERROR     0x02 // bit 1 mask
#define DATA_READY        0x01 // bit 0 mask

void foo(void)
{
	uint8_t reg = read_line_register();
	if (reg & DATA_READY) {
		// Data is ready
	}
}
```

Traditional approach

- Traditional approach works very well for bit-wise access
- For multi-bit access, use **bit-fields**

A **bit-field** is a data structure that allows access and/or operation of individual bits or group of bits of a word

Bit-fields in C/C++

- Declared as a struct
  - Each member is a bit-field within a word
  - Accessed like members of a struct
  - Fields may be named or un-named

```
struct structure_tag {
        typeA memberA1 : bit_widthA1;
        typeA memberA2 : bit_widthA2;
        ...
        typeB memberB1 : bit_widthB1;
        typeB memberB2 : bit_widthB2;
        ...
} variable_list;
```

Bit-fields example (serial port line reg.)

```
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};

void foo(void)
{
        struct sp_line_reg reg = read_line_register();
        if (reg.data_ready) {
                // Data is ready
        }
}
```

Bit-fields example (serial port line reg.)

```
struct sp_line_reg {
    uint8_t fifo_error    : 1;
    uint8_t empty_dhr     : 1;
    uint8_t empty_thr     : 1;
    uint8_t break_intr    : 1;
    uint8_t framing_error: 1;
    uint8_t parity_error : 1;
    uint8_t overrun_error: 1;
    uint8_t data_ready    : 1;
};
```

```
struct sp_line_reg {
    uint8_t fifo_error    : 1,
            empty_dhr     : 1,
            empty_thr     : 1,
            break_intr    : 1,
            framing_error: 1,
            parity_error : 1,
            overrun_error: 1,
            data_ready    : 1;
};
```

Bit-fields example (vs traditional)

```
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};

void foo(void)
{
    struct sp_line_reg reg =
        read_line_register();
    if (reg.data_ready) {
        // Data is ready
    }
}
```

```
#define FIFO_ERROR      0x80 // bit 7 mask
#define EMPTY_DHR       0x40 // bit 6 mask
#define EMPTY_THR       0x20 // bit 5 mask
#define BREAK_INTR      0x10 // bit 4 mask
#define FRAMING_ERROR   0x08 // bit 3 mask
#define PARITY_ERROR    0x04 // bit 2 mask
#define OVERRUN_ERROR   0x02 // bit 1 mask
#define DATA_READY      0x01 // bit 0 mask

void foo(void)
{
        uint8_t reg =
            read_line_register();
        if (reg & DATA_READY) {
                // Data is ready
        }
}
```

Problems with bit-fields

- The actual arrangement of the bits on memory depends on the compiler and/or “endian-ness” of the CPU
- **Bit-fields are not portable**

```
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};
```

Can be mapped in memory this way:

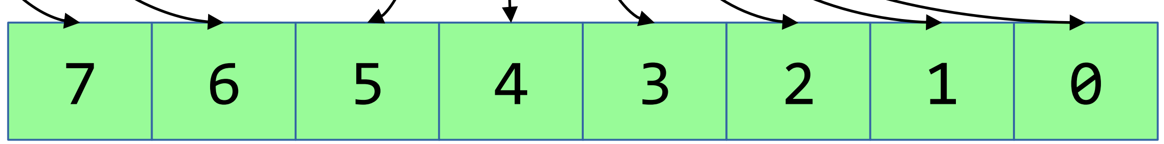
7	6	5	4	3	2	1	0

Problems with bit-fields

- The actual arrangement of the bits on memory depends on the compiler and/or “endian-ness” of the CPU
- **Bit-fields are not portable**

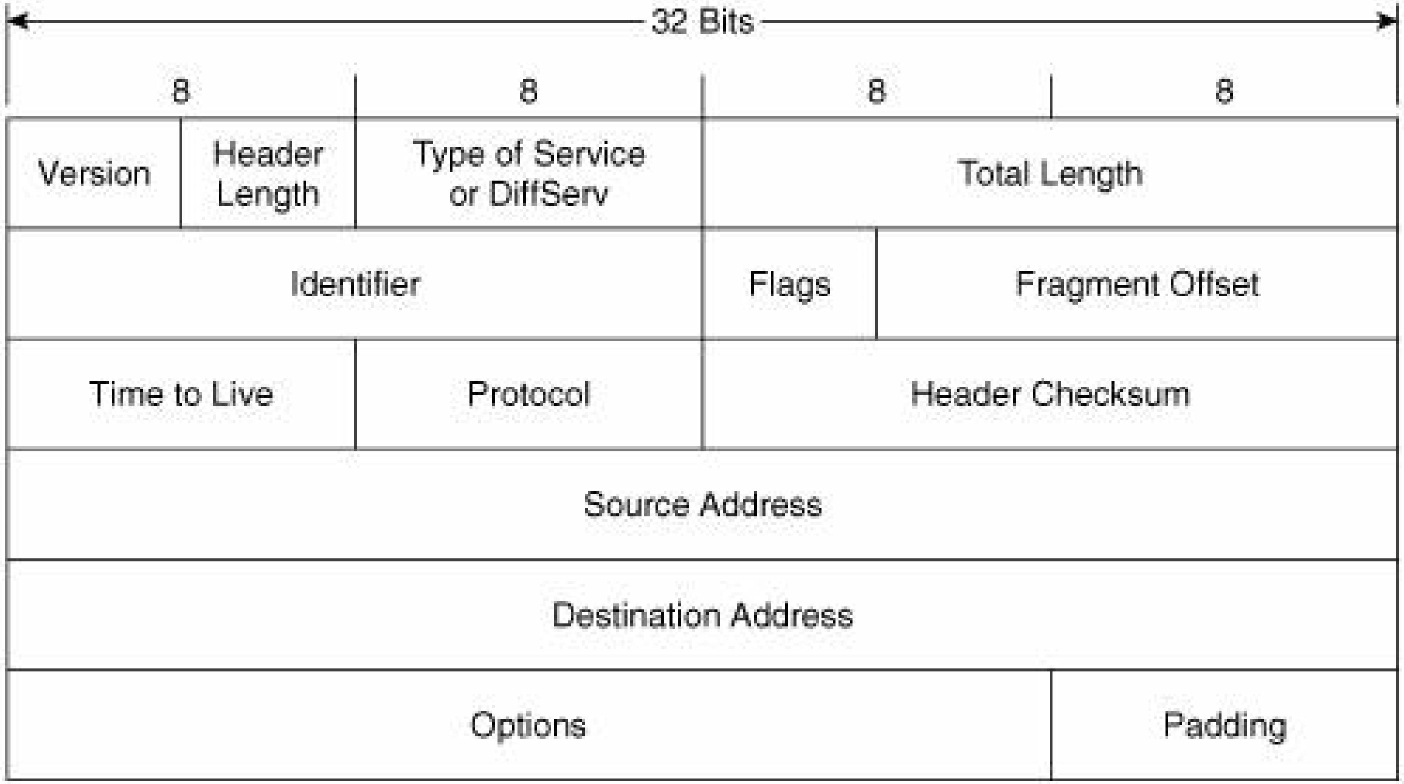
Or possibly this way:

```
struct sp_line_reg {
        uint8_t fifo_error    : 1;
        uint8_t empty_dhr     : 1;
        uint8_t empty_thr     : 1;
        uint8_t break_intr    : 1;
        uint8_t framing_error: 1;
        uint8_t parity_error : 1;
        uint8_t overrun_error: 1;
        uint8_t data_ready    : 1;
};
```

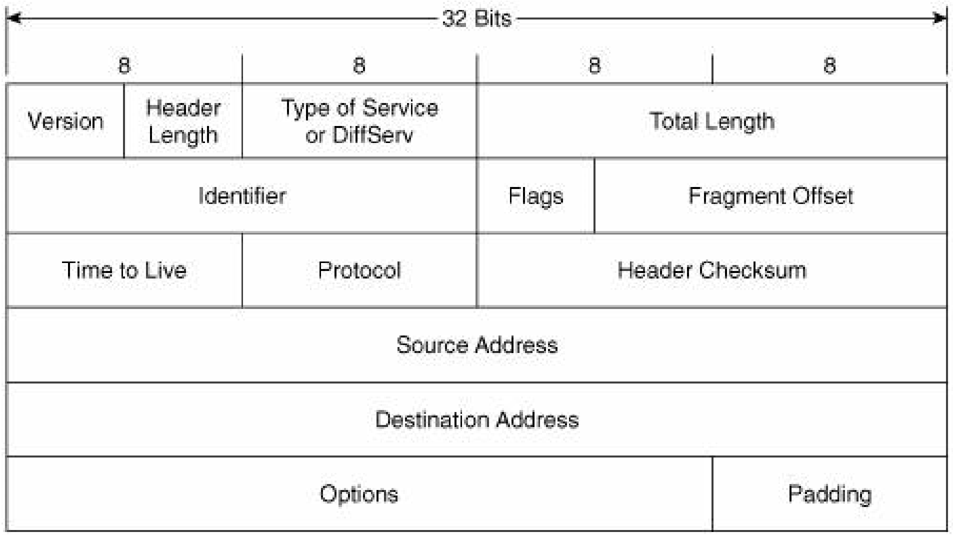


Bit-fields example: IPv4 packet header

Internet Protocol version 4 packet header format:



Bit-fields example: IPv4 packet header




```
struct iphdr {
    #if defined(__LITTLE_ENDIAN_BITFIELD)
        uint8_t ihl    :4,
                version:4;
    #elif defined (__BIG_ENDIAN_BITFIELD)
        uint8_t version:4,
                ihl    :4;
    #else
        #error  "Please fix <asm/byteorder.h>"
    #endif
         uint8_t   tos;
         uint16_t  tot_len;
         uint16_t  id;
         uint16_t  frag_off;
         uint8_t   ttl;
         uint8_t   protocol;
         uint16_t  check;
         uint32_t  saddr;
         uint32_t  daddr;
};
```

Bit-fields example: IPv4 packet header

```
void ipv4_receive(void *pkt, uint32_t my_addr)
{
    struct iphdr *iph = (struct iphdr *) pkt;

    if(iph->version != 4) {
       // Incorrect version, return
       return;
    }

    if(iph->daddr == my_addr) {
       // This packet is for me
       // Do stuff to receive it!
    }
}
```

Memory alignment

- Many computer systems place restrictions on the allowable addresses for the basic data types
  - The address for some type must be a multiple of some value k (typically 2, 4, or 8)

- Such **alignment restriction** is meant for improving memory access performance

Linux memory alignment policy on 32-bit x86

- 1-byte data types (e.g., int8_t, uint8_t, char) can have any address
- 2-byte data types (e.g., int16_t, uint16_t, short) must have an address that is a multiple of 2
- Larger data types (e.g., int32_t, int64_t, float, double) must have an address that is a multiple of 4

Memory alignment on structures

- Compiler may need to insert gaps in the field allocation to ensure that each structure element satisfies its alignment requirement

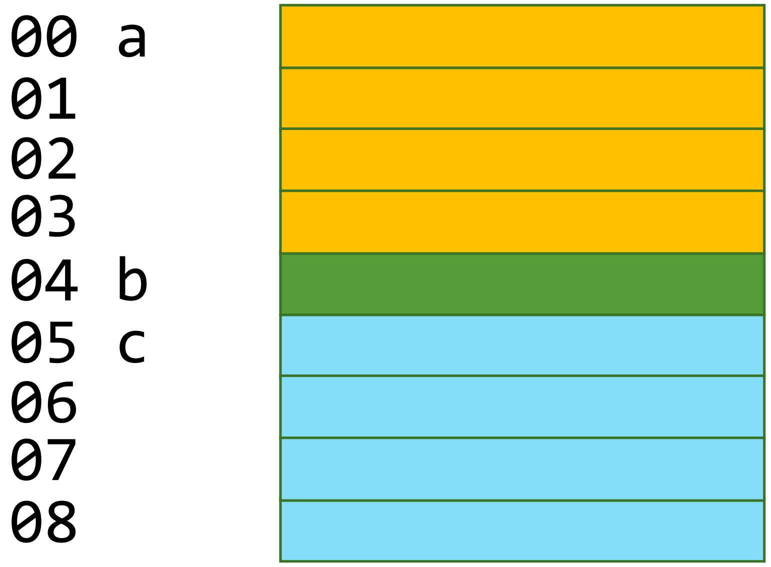
- The entire structure then has to follow a required alignment for its starting address

Memory alignment on structures

- Consider the following:

```
struct s1 {
    int32_t a;
    int8_t b;
    int32_t c;
};
```

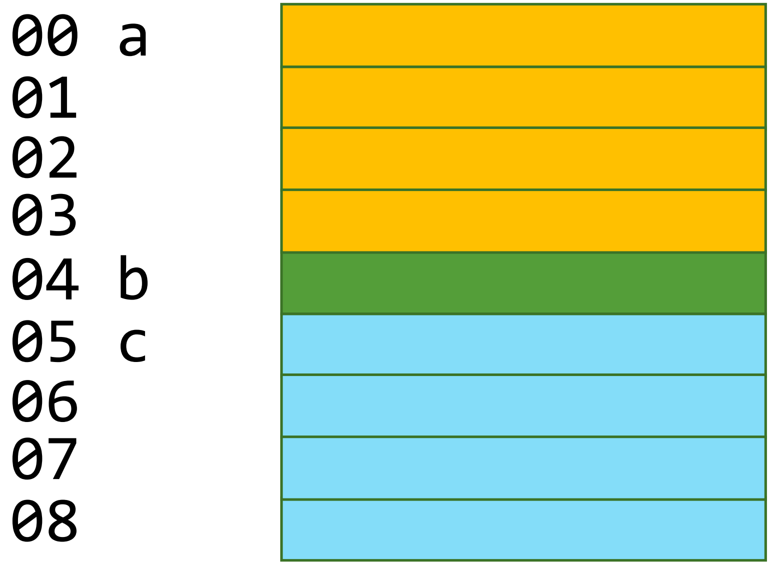
- Minimally, `s1` can be allocated in memory:



- Member c is not aligned!
  - Its address should be a multiple of 4

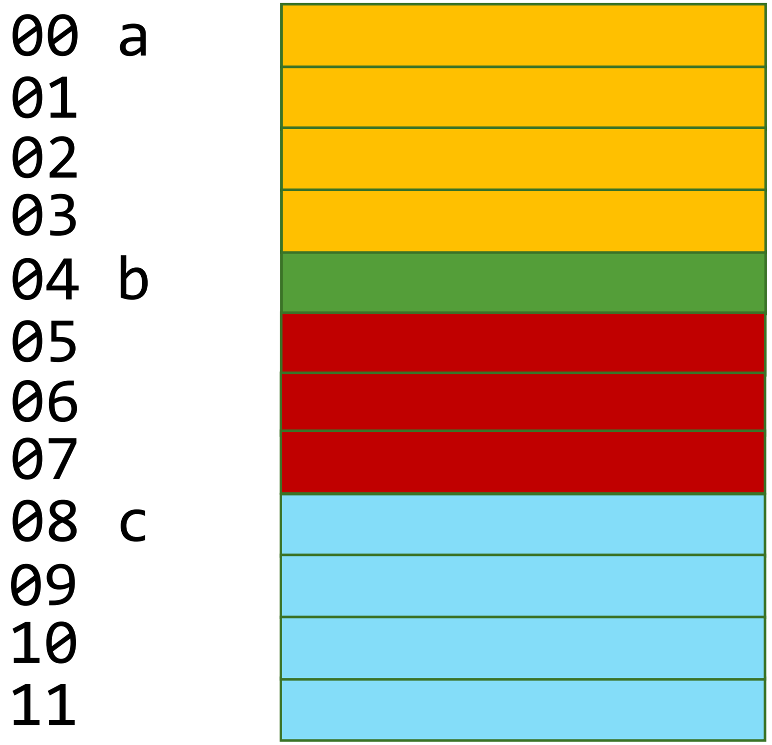
Structure padding

- Minimally, `s1` can be allocated in memory:



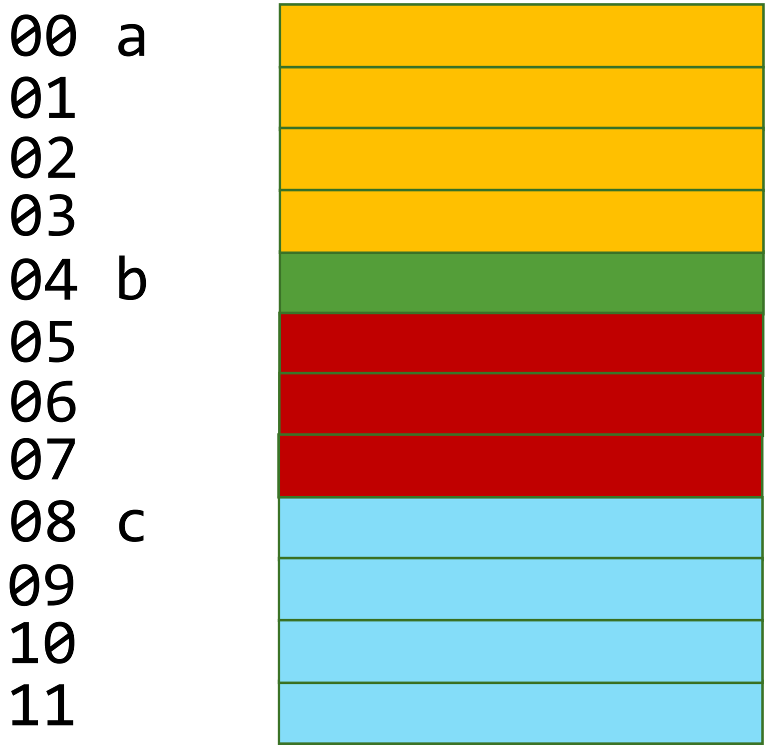
- Member c is not aligned!
    - Its address should be a multiple of 4

- Solution: Compiler will add **padding** to align members



What else?

- Solution: Compiler will add **padding** to align members



- In addition to padding some of members, the structure itself should begin at an address that is a multiple of 4

Structure packing

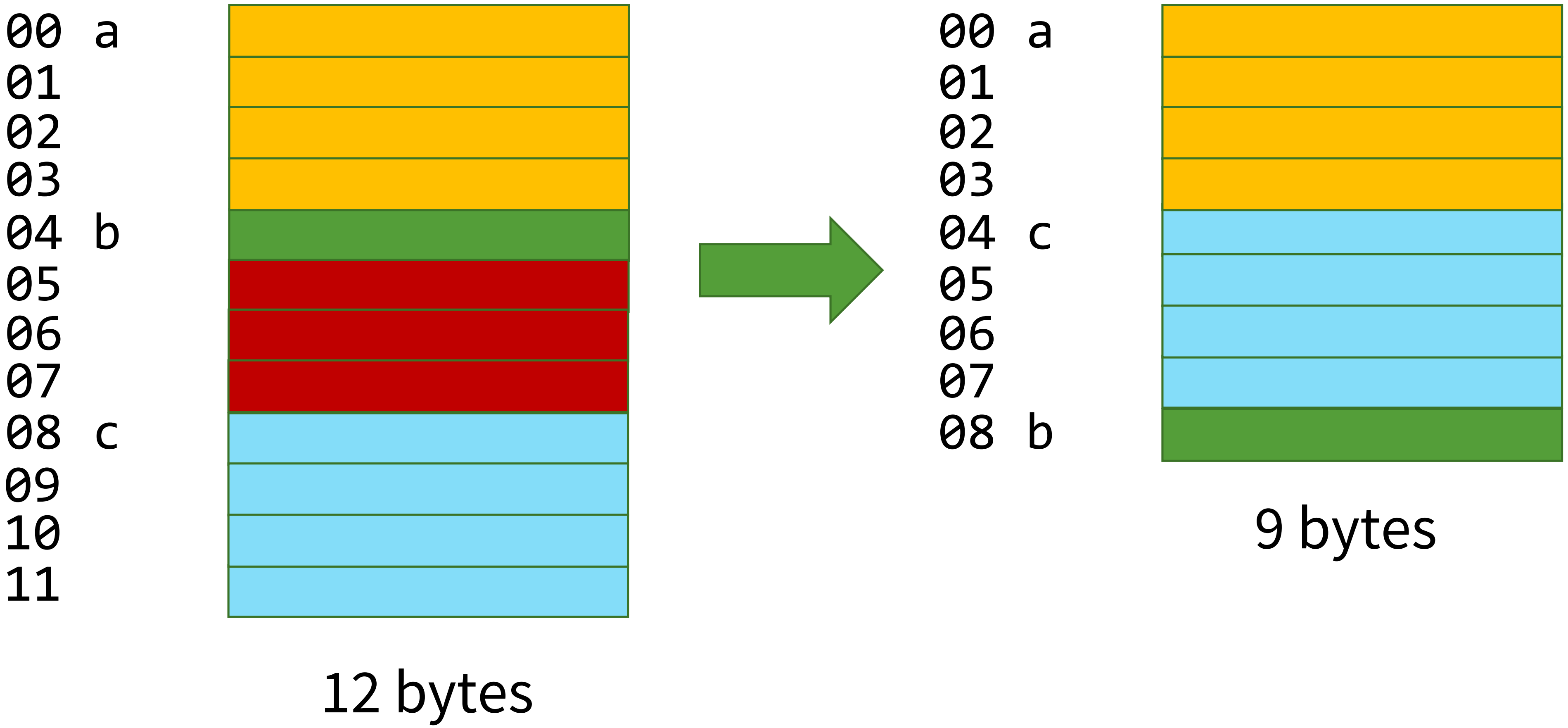
- Packing: re-arranging structure members to reduce waste due to padding for data alignment

```
struct s1 {
    int32_t a;
    int8_t b;
    int32_t c;
};
```

→

```
struct s1 {
    int32_t a;
    int32_t c;
    int8_t b;
};
```

Structure packing



Overriding alignment

- GCC and G++ provides an option to override alignment restrictions on structures
  - May not work on all CPUs
  - Will result in slower code

```
struct s1 {
    int32_t a;
    int8_t b;
    int32_t c;
};
```

```
struct s1 {
    int32_t a;
    int32_t c;
    int8_t b;
} __attribute__((packed));
```

Next lecture

- Process management