# Bytes, bits, etc.,

**R. Inkulu**
**http://www.iitg.ac.in/rinkulu/**

# Outline

1 Self-alignment

2 Bitwise operators

3 Bit-fields

4 Endianness

# Self-alignment of primitive types

```
int i; short s; char c; long l;

printf("%d, %d, %d, %d, %d\n", sizeof(i), sizeof(s),
 sizeof(c), sizeof(l));
//prints 4, 2, 1, 8

printf("%p, %p, %p, %p, %p\n", &i, &s, &c, &l);
//prints 0xbfdf07cc, 0xbfdf07ca, 0xbfdf07c9, 0xbfdf07d8
```

- *self-aligned*: value $v$ of a primitive type $typeA$ is stored starting from only bytes whose address is an non-negative integer multiple of $sizeof(typeA)$
- modern processor architectures' are designed to access addresses that are self-aligned efficiently; hence, compilers generate binary code to exploit the same

# Self-alignment of structures

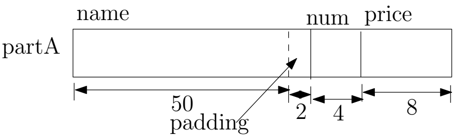
```
typedef struct {
   char *name;
   int num;
   double price;
} Part;

int main(void) {
   Part partA;
   printf("%d, %p\n", sizeof(partA), &partA);
     //prints 16, 0xbf88bf10
}
```

- like primitive typed values, custom objects are also self-aligned: object $o$ of a custom type (structure) $T$ can only be stored starting from bytes whose address is a non-negative integer multiple of the most restrictive member of $T$[1]

[1] helps in calculating internal and trailing paddings (see below)

# Internal padding for self-alignment




for any non-negative integer $y$, there must exist a non-negative integer $x$ such that

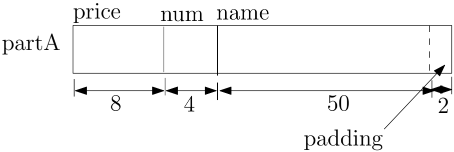
$4x = 8y + (50 + \delta)$; hence the padding $\delta$ before *num*

```
typedef struct {
   char name[50];
   int num;
   double price; } Part;
```

```
int main(void) {
    Part partA;
    printf("%d, %d, %d\n", sizeof(Part), (void*)&partA.num-(void*)partA.name,
     (void*)&partA.price-(void*)&partA.num); }
```

- padding is need for the purpose of member self-alignment (considering the self-alignment of the objects instantiated from that structure)

# Trailing padding for self-alignment




for any non-negative integer $y$, there must exist a non-negative integer $x$ such that

$8x = 8y + (62 + \delta)$; hence the padding $\delta$ at the end

```
typedef struct {
double price;
int num;
char name[50]; } Part;
```

```
int main(void) {
    Part partA;
    printf("%d, %d, %d\n", sizeof(Part), (void*)&partA.num-(void*)&partA.price,
     (void*)partA.name-(void*)&partA.num);
}
```

- trailing padding is needed to take care of defining array of structure objects
- however, leading padding is not allowed (according to the standard, address of first member of a structure object must need to be same as the address of object itself)

# Examples

- typedef struct{ short, char} A;
- typedef struct{ char*, int} B;
- typedef struct{ char*, short, int } C;
- typedef struct{ int, char, short} D;
- typedef struct{ int, char [3], short [10]} E;
- typedef struct { C, E [6], D [10]} F; [2]
- try more ...

homework: analyze the sizes and draw the memory layouts

homework: analyze for the best possible ordering of structure members in the structure (based on their sizes)

[2] recursively apply the self-alignment rules

# Outline

1 Self-alignment

2 Bitwise operators

3 Bit-fields

4 Endianness

# Bitwise operators

- bitwise AND ’&’
- bitwise OR ’|’
- bitwise XOR ’ˆ’
- bitwise NOT ’∼’
- right shift ’>>’
- left shift ’<<’

# Few examples

```
unsigned int i = 23;

//sets the j-th LSB of i
i = i | (1 << j-1);
//clears the j-th LSB of i
i = i & ~(1 << j-1);
//toggling the j-th LSB of i
i = i ^ (1 << j-1);

//multiplies i by 2 power j
i = i << j;
//divides i by 2 power j
i = i >> j;
//i modulo 32
unsigned int r = i & 0x1F;
//is i a power of 2
if (i & i-1 == 0) return 1;
```

# Bit-masks (using macros)

in maintaining a *symbol table*, a compiler wants to categorize each identifier and/or determine the kind:

```
#define KEYWORD 01
#define EXTRENAL 02
#define STATIC 04

unsigned int flags;
...
flags |= EXTERNAL | STATIC;
  //turns on the EXTERNAL and STATIC bits in flags

flags &= ~(EXTERNAL | STATIC);
  //turns off the EXTERNAL and STATIC bits in flags

if ((flags & (EXTERNAL | STATIC)) == 0) ...
  //evalutes to true if both bits are off
```

# Bit-masks (using enum)

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };

unsigned int flags;
...
flags |= EXTERNAL | STATIC;
  //turns on the EXTERNAL and STATIC bits in flags

flags &= ~(EXTERNAL | STATIC);
  //turns off the EXTERNAL and STATIC bits in flags

if ((flags & (EXTERNAL | STATIC)) == 0) ...
  //evalutes to true if both bits are off
```

# Outline

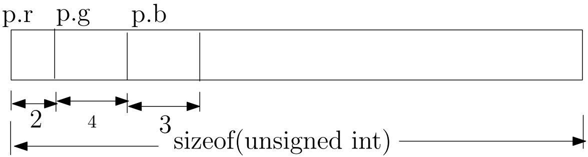

1 Self-alignment

2 Bitwise operators

3 Bit-fields

4 Endianness

# Intro to bit-fields




(field assignment is implementation-specific; this fig shows one such possibility)

```
struct Permissions {
  unsigned int r : 2;
    //only int, unsigned int types are allowed
  unsigned int g : 4;
  unsigned int u : 3;
};
struct Permissions p;   //sizeof(p) is 4
```

- contiguous set of adjacent bits is termed as a *bit-field* (in the above example, r, g and u are bit-fields)
- helps in saving space

# Motivation: bit-masks using bit-fields

```
struct {
  unsigned int isKeyword : 1;
  unsigned int isExtern : 1;
  unsigned int isStatic : 1
} flags;

flags.isExtern = flags.isStatic = 1;
  //turns on both the bits

flags.isExtern = flags.isStatic = 0;
  //turns off both the bits

if (flags.isExtern == 0 && flags.isStatic == 0) ...
  //evalutes to true if both bits are off
```

- now the code is clean: bit-level optimization is not in the code instead it is hidden in the struct; together with space-efficiency

# Space allocation in structures with bit-fields

```
struct Permissions {
  unsigned int r : 2;
  unsigned int g : 4;
  unsigned int u : 3;
};
int main(void) {
  struct Permissions p;
  printf("%d\n", sizeof(p));
    //prints 4
}
```

- allocates sizeof(unsigned int) and packs bit-fields into it successively until it is full or padding is enforced; again, allocates another sizeof(unsigned int) etc.,[3]
- bit-fields do not have addresses (hence, & operator not applicable)

[3] if the prior members are not a bit field but some other type whose size is smaller than unsigned int, then it allocates sizeof(unsigned int) to start with; when a member is not a bit-field, it utilizes the space allocated if it can accommodate it

# Padding for alignment

```
struct Permissions {
  unsigned int r : 2;
  unsigned int g : 4;
  unsigned int u : 3;
  unsigned int : 0;     //for padding
  unsigned int x : 5;
};

int main(void) {
  struct Permissions p;
  printf("%d\n", sizeof(p));
    //prints 8
}
```

- unnamed fields are used for padding; special width 0 is used to force alignment at the next unsigned int boundary (next field will be stored in a new unsigned int)

# Outline


1. Self-alignment
2. Bitwise operators
3. Bit-fields
4. Endianness

# Description

Convention used in storing (and interpreting) bytes making an object of any primitive data type is known as *endianness*.

# Two typical endian formats

little-endian representation

| 00001101 | 00001100 | 00001011 | 00001010 |
|---|---|---|---|
| $p$ | $p+1$ | $p+2$ | $p+3$ |

big-endian representation

| 00001010 | 00001011 | 00001100 | 00001101 |
|---|---|---|---|
| $p$ | $p+1$ | $p+2$ | $p+3$ |

$p$ is the address of first byte in which an unsigned int is stored

an unsigned int: $00001010\,00001011\,00001100\,00001101_{(2)}$
(or $0x0A0B0C0D$, or $168496141_{(10)}$)

- *little-endian* machine: least significant byte of a value is stored in the smallest addressed byte of the word; ex. Intel x86 machines

- *big-endian* machine: most significant byte of a value is stored in the smallest addressed byte of the word; ex. Motorola 6800 machines

# Testing endianness

```
unsigned int i = 1;
char *q = (char*)&i;

if (((int) q[0]) == 1)
     printf("little-endian\n");
else
     printf("big-endian\n");
//prints little-endian
```

# Converting little-endian to big-endian (and vice versa)

```
unsigned int value = 0x0A0B0C0D, result = 0;
printf("%d bytes\n", sizeof(unsigned int));
//prints 4 bytes
char *p = (char*) &value;
printf("%d: %d, %d, %d, %d\n",
       value, p[0], p[1], p[2], p[3]);
//prints 168496141: 13, 12, 11, 10

result = (value & 0x000000FF) << 24;
result |= (value & 0x0000FF00) << 8;
result |= (value & 0x00FF0000) >> 8;
result |= (value & 0xFF000000) >> 24;
p = (char*)&result;
printf("%d: %d, %d, %d, %d\n",
       result, p[0], p[1], p[2], p[3]);
//prints 218893066: 10, 11, 12, 13
```

homework: directly modify the contents of value while using a byte of temporary variable

# Setting a bit value in unsigned int

```
unsigned int i = 0x1;
char *p = (char*) &i;
printf("%d: %d, %d, %d, %d\n",
       i, p[0], p[1], p[2], p[3]);
    //prints 1: 1, 0, 0, 0

p[1] |= 0x08;
   //sets fourth least-significant bit in p[1]
printf("%d: %d, %d, %d, %d\n",
       i, p[0], p[1], p[2], p[3]);
   //prints 2049: 1, 8, 0, 0
```

# Binary representation of numbers

- short, int, unsigned int
- ASCII values of chars
- float[4], double

homework:

* given a double $d$, find its equivalent binary representation $b$
* convert $b$ to the relevant endian format of your machine

[4] IEEE 754 standard: http://www.h-schmidt.net/FloatConverter/IEEE754.html

# Two's complement of binary numbers

For any negative number $n$, the two's complement of $n$ is the ones' complement of $n$ added with one; for positive numbers, two's complement is same as that number itself. Advantages:

- fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers
- zero has only a single representation
- no need to examine the signs of the operands to determine when doing addition, subtraction, and/or multiplication of numbers

based on the need, a number's two's complement is computed (with the help of hardware) just before doing the algebra in registers