CSC 411

Computer Organization (Fall 2024)
Lecture 5: Casting, Byte ordering, Pointers

Prof. Marco Alvarez, University of Rhode Island

Quick notes

- For Windows users requiring Linux access:
  - install a Linux distribution
    - standalone or side-by-side with Windows (dual-boot)
  - use Windows subsystem for Linux (WSL)
    - https://docs.microsoft.com/en-us/windows/wsl/install-win10
  - use a virtual machine

Practice

- What decimal value is this bit sequence representing?

```
1 0 0 1 0
```

  - assume bit-width $n = 5$, unsigned
  - assume bit-width $n = 5$, two's complement
  - assume bit-width $n > 5$

Practice

- Assume $n = 8$ and two's complement
  - what is the largest positive?
  - what is the smallest positive?
  - what is the largest negative? (farther from 0)
  - what is the smallest negative? (closer to 0)

Casting integer values

Casting in C

- Constants
  - considered signed integers by default, use U suffix for unsigned, e.g. `502123U`
- Casting
  - changes interpretation, the bit sequence is **maintained**
  - NOT the same as converting a positive value d into its negative $-d$
  - two's complement conversions between signed and unsigned maintain the same bit sequence, interpretation changes
- Types of casting
  - **explicit casting:** requires parenthesized type specification
  - **implicit casting:** occurs automatically in assignments and function calls
    - e.g. assigning `unsigned int` to `int`

Casting to larger data types

- Sign extension
  - transform n-bit integer to m-bit integer ($m > n$), preserving value
  - method: make $m - n$ copies of the MSB

				−8	4	2	1	
				0	1	1	0	6
−128	64	32	16	8	4	2	1	
0	0	0	0	0	1	1	0	6
				−8	4	2	1	
				1	1	1	0	−2
−128	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	0	−2

Practice

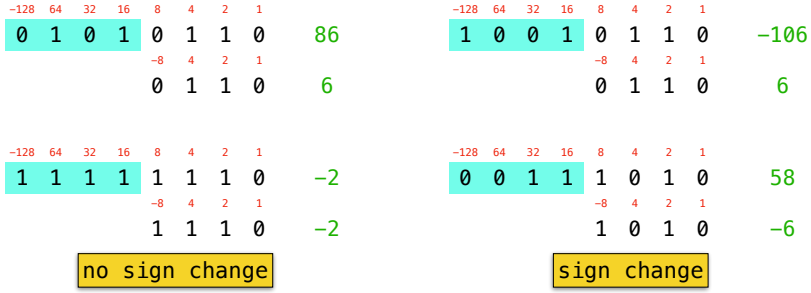
- What is the decimal value of this 64-bit number represented using two's complement?

`0xFFFFFFFFFFFFFFF8`

`11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111000`

Casting to smaller data types

- Truncation
  - transform n-bit integer to m-bit integer ($m < n$), preserving rightmost m bits
  - method: drop top $n - m$ bits



What is the output?

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

int main() {
    // 411 is 0x019B
    // -411 is 0xFE65
    int16_t var1 = -411;
    uint16_t var2 = (uint16_t) var1;
    // %h (short integer)
    printf("%hd %x\n", var1, var1);
    printf("%hu %x\n", var2, var2);

    return 0;
}
```

```
malvarez — -zsh — 42×5
$ gcc -g casting.c -o prog
$ ./prog
-411 FE65
65125 FE65
$
```

Casting in expressions

- If an expression contains signed and unsigned integers …
  - signed values are **implicitly cast** to **unsigned**
    - **recall -2147483648 is the most negative number in 32-bit signed integers**

Expression	Type	Evaluation
0 == 0U	unsigned	true
-1 < 0	signed	true
-1 < 0U	unsigned	false
2147483647 > -2147483647 - 1	signed	true
2147483647U > -2147483647 - 1	unsigned	false
2147483647 > (int)2147483648U	signed	true
-1 > -2	signed	true
(unsigned)-1 > -2	unsigned	true

What is the output?

```c
#include <stdio.h>

int main() {
    char a = 254;
    unsigned char b = 254;
    unsigned int c = 0;

    printf("%d %d\n", a, b);

    if (-1 < c) {
        printf("yay\n");
    } else {
        printf("!!!???\n");
    }
}
```

```
Desktop — -zsh — 46×12
$ gcc expr.c -o prog
expr.c:4:14: warning: implicit conversion from
 'int' to 'char' changes value from 254 to -2
[-Wconstant-conversion]
    char a = 254;
         ~   ^~~
1 warning generated.
$ ./prog
-2 254
!!!???
$
```

Memory organization

Memory organization

- Memory as a **byte array**
  - used to store **data and instructions** for computer programs
  - contiguous sequence of bytes
  - each byte individually accessed via a **unique address**
- Memory address
  - **unique** numerical identifier for each byte in memory
  - **pointer** variables store memory addresses
  - provides indirect access to data stored at that location

Memory organization

- Data representation in memory
  - variables stored as byte sequences
  - interpretation depends on type
    - integers, floating-point numbers, characters, etc.
- OS provides private address space to each **"process"**
  - process: a program being executed
  - address space: enormous arrays of bytes visible to the process
  - typically implemented through virtual memory

Byte ordering

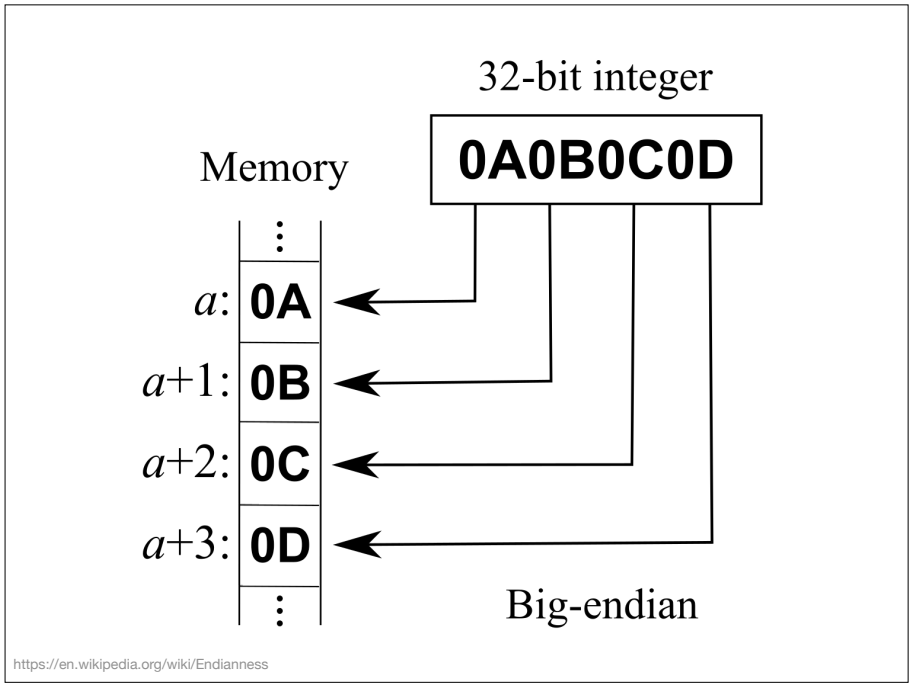
```
x = 0x1A2B3C4D
assume &x is 0x010
```

- Big Endian
  - stores the **most significant byte** at lowest memory address
  - IBM PowerPC, Motorola 68000, SPARC, used in network byte order

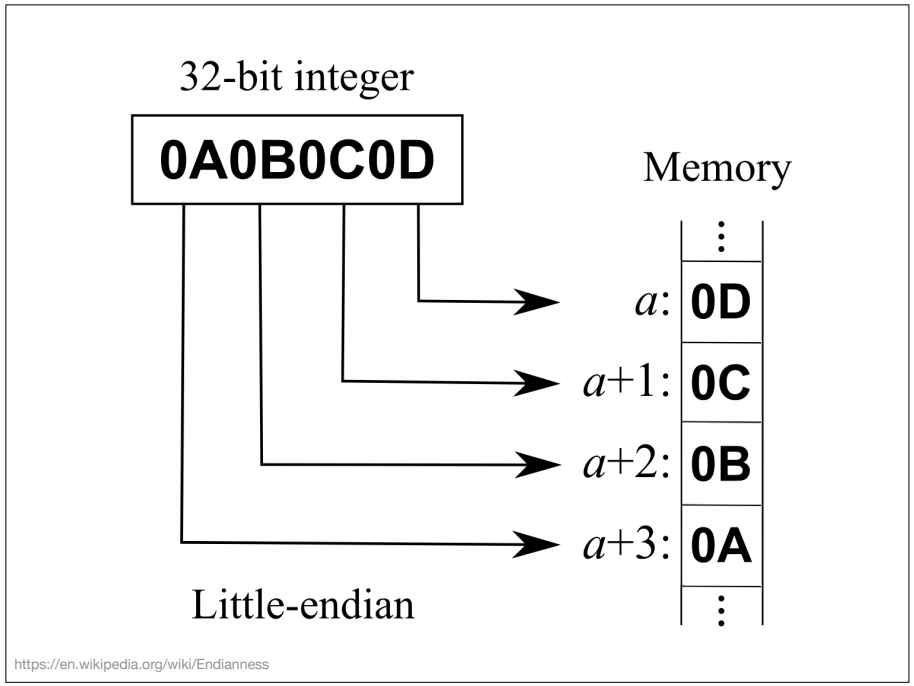
0x00D	0x00E	0x00F	0x010	0x011	0x012	0x013	0x014	0x015	0x016
			1A	2B	3C	4D			

- Little Endian
  - stores the **least significant byte** at lowest memory address
  - intel x86, ARM, RISC-V, MIPS

0x00D	0x00E	0x00F	0x010	0x011	0x012	0x013	0x014	0x015	0x016
			4D	3C	2B	1A			



https://en.wikipedia.org/wiki/Endianness



https://en.wikipedia.org/wiki/Endianness

Machine words

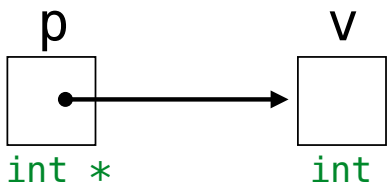
- Computers have a **"word size"**
  - usually the size of integer-valued data and memory addresses
    - 32-bit word size: address range of $0 \ldots 2^{32} - 1$
      - 4294967296 bytes or ~4GB
    - 64-bit word size: address range of $0 \ldots 2^{64} - 1$
      - 18446744073709551616 bytes or ~18EB
- Machines support multiple data formats
  - fractions or multiples of word size

Pointers

Variables and pointers

- Every variable exists at a **memory address** (regardless of **scope**)
  - memory address corresponds to a unique location
- The compiler translates names to addresses when generating machine code
  - C allows direct manipulation of variables and addresses

A **pointer** is a variable that stores the address of another variable



Pointers in C

- Must be declared before use
  - **pointer type** must be specified
- Pointer operators
  - **address-of operator**: get memory address of variable/ object

&

  - **dereference operator**: get value at given memory address

*

Null pointers and arrays

- The `null` pointer (`0x00000000`)
  - represents the absence of value
  - reading/writting with a null pointer can generate a **segmentation fault** signal
- Pointers and arrays
  - arrays in C decay to pointers (to the first element) in most contexts, but they are not themselves pointers
  - array names provide the address of the first element, can't be treated as variables

Declaring pointers

```c
// can declare a single
// pointer (preferred)
int *p;

// can declare multiple
// pointers of the same type
int *p1, *p2;

// can declare pointers
// and other variables too
double *p3, var, *p4;
```

Pointer operators

```c
int main() {
    int var = 10;
    int *ptr;
    ptr = &var;
    *ptr = 20;

    // ...

    return 0;
}
```

32-bit words

Address	Value	Variable
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		

Pointer operators

```c
int main() {
    int temp = 10;
    int value = 100;
    int *p1, *p2;

    p1 = &temp;
    *p1 += 10;

    p2 = &value;
    *p2 += 5;

    p2 = p1;
    *p2 += 5;

    return 0;
}
```

32-bit words

Address	Value	Variable
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		

Pointers and functions

```c
void increment(int *ptr) {
    (*ptr) ++;
}

int main() {
    int var = 10;

    increment(&var);
    increment(&var);

    // ...

    return 0;
}
```

32-bit words

Address	Value	Variable
…		
0x91340A08		
0x91340A0C		
0x91340A10		
0x91340A14		
0x91340A18		
0x91340A1C		
0x91340A20		
0x91340A24		
0x91340A28		
0x91340A2C		
0x91340A30		
0x91340A34		
…		

Pointer arithmetic

- Can add values to pointers
  - treats addresses as unsigned integers
- Must be careful !
  - p+1 adds the size of pointed variable
  - p+1 does NOT add 1 “byte”
- Can use pointer arithmetic for array traversal

`a[i]` is equivalent to `*(a+i)`

Advanced pointer concepts

- Declaring a pointer allocates space for the pointer
  - 4 (32-bit architecture) or 8 bytes (64-bit architecture), NOT additional memory
- Generic pointers (`void *`)
  - use carefully to avoid bugs/vulnerabilities
  - lack type checking, which can lead to errors if not managed
- Pointer to functions
  - allow indirect function calls

```
// declare and initialize
int (*func) (int, int) = &my_func;
// use
c = (*func)(a, b);
```

Changing a pointer inside a function

```
#include <stdio.h>

void seek(int *array, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (*array == key) {
            return;
        }
        array ++;
    }
}

int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(data, 3, 5);
    printf("%d\n", *p);

    return 0;
}
```

does it work?

Using double pointers

```
// function to search for a key in an array
// - pointer to an array of integers
// - an integer key
// - an integer n, the number of elements

void seek(int **p, int key, int n) {
    for (int i = 0 ; i < n; i++) {
        if (**p == key) {
            return;
        }
        (*p) ++;
    }
}
```

Using double pointers

```
int main() {
    int data[] = {1, 2, 3, 4, 5};
    int *p = data;

    seek(&p, 3, 5);
    printf("%d\n", *p);

    return 0;
}
```

Python Tutor: Visualize code in Python, JavaScript, C, C++, and Java

C (C17 + GNU extensions)
known limitations

```
 6 // - an integer key
 7 // - an integer n, the number of elements in the
 8 void seek(int **p, int key, int n) {
 9     for (int i = 0 ; i < n; i++) {
10         if (**p == key) {
11             return;
12         }
13         (*p) ++;
14     }
15 }
16
17 int main() {
18     int data[] = {1, 2, 3, 4, 5};
19     int *p = data;
20
21     seek(&p, 3, 5);
22     printf("%d\n", *p);
23
24     return 0;
25 }
```

Edit this code

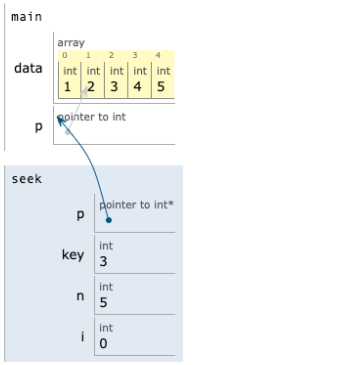
line that just executed
next line to execute

<< First | < Prev | Next > | Last >>

Step 9 of 17

Print output (drag lower right corner to resize)

Stack Heap



C/C++ details: none [default view]