Lecture 4:
Bit Operations and UTF-8 (Unicode!)

CSE 29: Systems Programming and Software Tools

Aaron Schulman (Shalev)

UCSDCSE
Computer Science and Engineering

Accessing individual bits in C data types

It is possible to access individual bits of integer data types in C

- char and int (as well as long and short modifiers)
- You can not do bit operations on floating point numbers (float and double)

There are special mathematical operators in C for doing operations on bits

- & - AND
- ^ - XOR
- | - OR
- >> - Shift Right
- << - Shift Left

How bit operations work: AND

```
char a = 0x1; // 00000001
char b = 0x5; // 00000101

// AND each bit of the two integers together
char a_and_b = a & b;
//   00000001
// & 00000101
// -----------
//   00000001

printf("%d\n", a_and_b); // What will the output be?
```

Masking: use case for bitwise AND

- **Masking:**
  - Selecting specific individual bits out of a binary representation of a number
  - Example:
    - » first_four_bits(**0b01010101**) = ?
    - » last_four_bits(**0b00000011**) = ?
    - » first_four_bits(**192**) = ?
    - » first_four_bits(**last_four_bits(_______)**);

How do we implement masking?

```
char first_four_bits( char c ) {
        return c & 0b11110000;
}

char last_four_bits( char c ) {
    return c & 0b00001111;
}
```

- Using the bitwise AND operator **&** we can select specific bit positions to view
- The bits selected come from the pattern of 1's that the variable is **&'ed** with

Demo: Masking used in practice!

- **Big Idea:**
  - Using masking, we can inspect (print) each individual bit in memory!
  - Doing this requires changing the mask over time

Let’s look at bit-by-bit masking in detail

var = 0b10010001

Mask	Place & mask	Print
0b10000000 = 128	0b100000000	1
0b01000000 = 64	0b000000000	0
0b00100000 = 32	0b000000000	0
0b00010000 = 16	0b00010000	1

We are ready to understand ASCII++

- How to handle the thousands of characters used in languages around the world?
  - ASCII does not define:
    - Spanish: **é**
    - Chineese: 中
    - Emoji: 🦀
- 256 bit patterns in one byte is not enough!
- Challenge: Millions of lines of code were written that assumed one byte ASCII chars

UTF-8: Introduced in at a conference in San Diego

Hello World
or
Καλημέρα κόσμε
or
こんにちは 世界

Rob Pike
Ken Thompson

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Plan 9 from Bell Labs has recently been converted from ASCII to an ASCII-compatible variant of Unicode, a 16-bit character set. In this paper we explain the reasons for the change, describe the character set and representation we chose, and present the programming models and software changes that support the new text format. Although we stopped short of full internationalization—for example, system error messages are in Unixese, not Japanese—we believe Plan 9 is the first system to treat the representation of all major languages on a uniform, equal footing throughout all its software.

Introduction

The world is multilingual but most computer systems are based on English and ASCII. The release of Plan 9 [Pike90], a new distributed operating system from Bell Laboratories, seemed a good occasion to correct this chauvinism. It is easier to make such deep changes when building new systems than by refitting old ones.

The ANSI C standard [ANSIC] contains some guidance on the matter of 'wide' and 'multi-byte' characters but falls far short of solving the myriad associated problems. We could find no literature on how to convert a *system* to larger character sets, although some individual *programs* had been converted. This paper reports what we discovered as we explored the problem of representing multilingual text at all levels of an operating system, from the file system and kernel through the applications and up to the window sys-

Solution: Bit flags!

- **Terminology… "Code Point":**
  - A code point is an integer representing a character (e.g., 65 == 'A')
- **Normal ASCII Code Point**: Highest order bit of byte is **0xxxxxxx**
- **Multi-byte Code Point**: Highest order bit of byte is **1xxxxxxx**
- **First byte dictates code point length:**
  - **11xxxxxx = 2 bytes**
  - **111xxxxx = 3 bytes**
  - **1111xxxx = 4 bytes**
- **Bytes in the middle (and end) start with:**
  - **10xxxxxx**

Code point construction from multi-byte

- **é =** $\boxed{\texttt{0b11}}$000011 $\boxed{\texttt{0b10}}$101001

2 bytes **continue**

000011101001 = 233

The Code Point

Extracting Code Point

```
int32_t code_point_2(char c1, char c2) {
        char part1 = c1 & 0b00011111;
        char part2 = c2 & 0b00111111;
        return (c1 * 64) + c2;
}

char eacc = ”é”; // UTF-8 char so it will be multiple bytes!
code_point_2(eacc[0], eacc[1]) = 233;
```