CSM CS61C Note #1: Number Representation, C Intro, Memory, Floating Point

Anthony Han

1 Number Representation

Number Representation

Within the computer, everything is a number ...

- of a fixed size:

  1 byte = 8 bits, 1 half word = 16 bits, 1 word = 32 bits, 1 double word = 64 bits, ...

- based on **bits** - binary digits.

Different ways to represent an integer. Take 123_{10} as an example:

- **Binary (2)**: $123_{10} = 1 \cdot 2_{10}^6 + 1 \cdot 2_{10}^5 + 1 \cdot 2_{10}^4 + 1 \cdot 2_{10}^3 + 1 \cdot 2_{10}^1 + 1 \cdot 2_{10}^0$
  $= 1 \cdot 10_2^6 + 1 \cdot 10_2^5 + 1 \cdot 10_2^4 + 1 \cdot 10_2^3 + 1 \cdot 10_2^1 + 1 \cdot 10_2^0 = 1111011_2.$
- **Octal (8)**: $123_{10} = 1 \cdot 8_{10}^2 + 7 \cdot 8_{10}^1 + 3 \cdot 8_{10}^0 = 1 \cdot 10_8^2 + 7 \cdot 10_8^1 + 3 \cdot 10_8^0 = 173_8.$
- **Decimal (10)**: $123_{10} = 1 \cdot 10_{10}^2 + 2 \cdot 10_{10}^1 + 3 \cdot 10_{10}^0 = 123_{10}.$
- **Hexadecimal (16)**: $123_{10} = 7 \cdot 16_{10}^1 + 11 \cdot 16_{10}^0 = 7 \cdot 10_{16}^1 + 11 \cdot 10_{16}^0 = 7B_{16}.$

Transformation between binary (2), octal (8), and hexadecimal (16) representations are relatively easy. If we want to transform FFF_{16} into binary representation, we divide and conquer. Given that $F_{16} = 1111_2$ and $10_{16} = 16 = 2^4 = 10_2^4$, we have

$$\begin{aligned} FFF_{16} &= F_{16} \cdot 10_{16}^2 + F_{16} \cdot 10_{16}^1 + F_{16} \cdot 10_{16}^0 \\ &= 1111_2 \cdot 10_2^8 + 1111_2 \cdot 10_2^4 + 1111_2 \cdot 10_2^0 = 1111\ 1111\ 1111_2. \end{aligned}$$

Similarly, if we want to transform a binary representation into a hexadecimal (or octal) representation, we divide the former into four-by-four segments (three-by-three for octal), transform each segment, and put all those together. This is also illustrated by the previous deduction, but the other way round.

Signed and Unsigned Integers

- **Signed integers** in C, C++, Java: for **data storage** purposes.

```c
int x = -7, y = 255, z;
```

- **Unsigned integers** in C, C++: for **addresses**.

  Unsigned integers in 32-bit word represent 0 to $2^{32} - 1 (\approx 4 \cdot 10^9)$.

Signed Integers and Two's Complement Representation

- Most significant (leftmost) bit is the **sign bit**: 0 meaning positive (including 0), 1 meaning negative.
- In two's complement, 32-bit word represents 2^{32} integers from -2^{31} to $2^{31}-1$.
- Two's complement is most ideal and is used by every computer today.
- In two's complement, **sign bit has negative weight.** E.g., for a 4-bit system, $-3_{10} = -8_{10} + 5_{10} = 1000_2 + 0101_2 = 1101_2$. Making two's complement this way, however, is inefficient.
- Alternatively, **to obtain two's complement for a negative number, invert all bits of its additive inverse, then add 1.**
- **Overflow occurs when magnitude of result is too big to fit into result representation.**
  - Examples of overflow of a 4-bit system: $7_{10} + 1_{10} = 0111_2 + 0001_2 = 1000_2 = -8_{10}$, $-8_{10} + -1_{10} = 1000_2 + 1111_2 = 10111_2 = 7_{10}$.
  - This can be resolved by monitoring carry bits on the most significant bit (MSB), as overflow **happens if and only if carry into MSB does not equal carry out MSB.**

As a sidenote, a collection of N bits can represent one of any 2^N possible things. A collection of 32 bits can be used to represent:

- A single unsigned number ($0 \sim 2^{32}-1$),
- Or a single signed number in two's complement ($-2^{31} \sim 2^{31}-1$),
- Or a 16-bit unsigned number, followed by a 10-bit signed integer, followed by 6 true/false bits.

2 C Intro

Compile vs. Interpret

- C compilers map C programs **directly** into *architecture-specific* machine code.
  - Java converts to *architecture-independent* bytecode that may be then compiled by a just-in-time compiler (JIT).
  - Python converts to a byte code *at runtime*.
- **Advantages:** excellent run-time performance with reasonable compile time.
- **Disadvantages:**
  - *Architecture-specific* - depends on processor type and operating system.
  - "Change → Compile → Run" cycle slow during development (though make only rebuilds changed parts).

C Pre-Processor (CPP)

- C source files first pass through macro processor/CPP, before compiler sees code.
- Replaces comments with a single space.
- Commands begin with '#'.

```c
#include "file.h"        // Inserts file.h into output
#include <stdio.h>       // Header files in standard location
#define PI (3.14159)     // Define constants with macros
#if / #endif             // Conditional inclusion of text
```

CPP Macros may cause errors, as it only changes the text of the program.

```c
#define twox(x) (x + x)
twox(y++);  // This is equivalent to (y++ + y++)
```

Typed Variables in C

- In C, you **must** declare the type of the variable at the beginning of a block and before use; the type **cannot change**. Variables not initialized will **hold garbage**, which may lead to undefined behavior.
- Some variable types are: int, unsigned int, float, double, char, long, long long.
- It is only guaranteed that sizeof(long long) $\geq$ sizeof(long) $\geq$ sizeof(int) $\geq$ sizeof(short) $\geq$ 16b and that sizeof(long) $\geq$ 32b.
- In C, **Only 0 and *NULL* evaluates to false.**

Consts and Enums in C

- Const is assigned a tixed value once in the declaration; cannot change during execution.

```c
const float golden_ratio = 1.618;
const int days_in_week = 7;
```

Can be of any type.

- Enums is a group of related integer constants.

```c
enum color {RED, GREEN, BLUE};  // Assigns 0 to RED, 1 to GREEN, 2 to
    BLUE
```

Typed Functions in C

- Must declare return value type - can be any variable type (including void).
- Must declare formal parameters type - analogously.

```c
int sum(int x, int y){
  return x + y;
}
```

- Syntax: **main** - called when C program starts

- C executable a.out is loaded into memory by OS.
- OS sets up stack, then calls into C runtime library.
- Runtime initializes memory and other libraries, then calls main function.

Now we have our first C program: Hello World!

```
#include <stdio.h>
int main(){
  printf("\nHello World\n");
  return 0;
}
```

Structs in C

Structs are structured groups of variables with **dot notation**.

```
typedef struct{
    int length;
    int year_recorded;
} Song;
Song song1;
song1.length = 213;
song1.year_recorded = 1994;
```

Unions in C

Unions represent a single piece of data in different ways.

- They provide enough space for the **largest element**.
- All members of the union change when one of them is modified.

```
union foo {
  int a;
  char b;
  union foo *c;
};
union foo f;
f.a = 0xDEADB33F;
  // Treat f as an integer and store that value.
```

sizeof()

Returns the number of **bytes** an object occupies, including padding needed for alignment (see §3). For instance,

- *sizeof(char) = 1*;
- *sizeof(int) = 4* on a 32b architecture.

Control Flow in C

- **if-else**

```c
if (<expression>) <statement>
else
  <statement>
```

- **while**

```c
while (<expression>) <statement>
```

do ... while

```c
do
        <statement>
while (<expression>);
```

- **for**

```c
for (<initialize>; <check>; <update>) <statement>
```

- **switch**

```c
switch (<expression>){
  case <const>:
    <statement>
    break;    // To avoid switch from running remaining cases; not mandatory
  default:
    <statement>
}
```

- **goto** - avoid using!

Operators in C

- Arithmetic: +, -, *, /, %
- Assignment: =
- Augmented assignment: +=, -=, etc.
- Bitwise: ˜, &, |, $\bigwedge$,<<, >>
- Boolean: !, &&, ||
- Equality: ==, !=
- Order relations: <, <=, >, >=
- Increment/Decrement: ++, –
- Member selection: ., ->
- Ternary operator: ? :

Pointers

- The C memory model is **arranged in bytes**; each byte has an **address**.
- A word is big enough to hold an address: word is 32b for 32b architecture, 64b for 64b architecture.
- An **address** refers to a particular memory location.
- A **pointer** is a variable that contains the address of a variable.
- Pointer syntax:

```c
int *p;
  // Tells the compiler that p is an address of an int
p = &y;
  // Tells the compiler to assign y's address to p
  // & is the address operator
z = *p;
  // Tells the compiler to assign value at p's address to z
  // * is the dereference operator
```

- Pointers can be used to pass variables that can be changed by a function of different scopes:

```c
void add_one(int *p) {
  *p = *p + 1;
}
int y = 3;
add_one(&y);
  // y is now 4
```

- Pointers can point to a specific kind of data:
  - *void ** can point to anything.
  - Type declaration determines how many bytes are fetched on each access through pointer - **Pointer arithmetic**. Use with caution!

```c
int *a = {0, 1, 2, 3, 4};
printf("%d\n", *(a + 2));  // This is equivalent to a[2]
  // Compiler realizes a is an int type pointer
  // and goes to <a's address> + 2 * sizeof(int)
```

  - A pointer can point to a pointer:

```c
int *p = &5;
int **q = &p;
  // q is a pointer that points to p,
  // which points to an integer 5 in the memory.
```

  - A pointer can even point to a function:

```c
int (*fn) (void *, void *) = &foo;
  // fn is a function taking two void * pointers and returning an
     int,
  // initially pointing to foo
  // (*fn)(x, y) will then call the function
```

- The **NULL** pointer - the pointer of all 0s:

  - Writing to or reading a null pointer would crash a program.
  - It is not hard to test for null pointers:

```
if (!p) { /* P is a null pointer */ }
if (q) { /* Q is not a null pointer */}
```

- Pointer dangers: declaring a pointer does **not** allocate space for the variable pointed to.

```
void f(){
  int *ptr;
  *ptr = 5; // Assigning value to garbage
}
```

- **Pointers and Structs - Arrow Notation**

```
typedef struct{
    int x;
    int y;
} Node;
Node *paddr;
int h = paddr -> x; // Arrow notation
int h = (*paddr).x; // Dot notation - not recommended!
```

- Advantages: convenient, allow cleaner and more compact code.
- Disadvantages: single largest bug source in C - dynamic memory management leads to dangling references and memory leaks.

3 Memory

C Arrays

- Declaration:

```
int ar[] = {7, 9};
  // Declare and initialize a 2-element int array.
int ar[2];
  // Declare a 2-element int array; NOT initialized.
  // # of arrays must be static: const int size is NOT supported in C!
```

- Arrays are **essentially pointers to the** 0^{th} **element.**
  - $\mathbf{a[i] \equiv *(a+i)}$. Still, bad style to interchange these two!
  - Arrays are passed into functions as pointers; the array size is lost! To deal with this problem:

```
int foo(int array[], unsigned int size){
  // Here, array size is input as a parameter.
  int i = 0;
  for (; i < size; i++)
    ...
}
```

  - Subtle difference: pointers can change address; arrays cannot.

- Arrays in C do not know its length, and bounds are unchecked! Hence, **use defined constants**:

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for (i = 0; i < ARRAY_SIZE; i++) { ... }
```

- **C strings** are just array of characters:

```
char string[] = "abc";
```

  - Last character followed by 0 or '\0' Byte (aka **null terminator**).
  - To output the length of the string (excluding the null terminator):

```
int strlen(char s[]){
  int n = 0;
  while (s[n] != 0)
    n++;
  return n;
}
```

```
int strlen(char s[]){
  int n = 0;
  while (*(s++) != 0)
    n++;
  return n;
}
```

```
int strlen(char s[]){
  char *p = s;
  while (*(p++) != 0)
    ; // Null body
  return (p - s - 1);
}
```

  (*strlen()* is in string.h)
  - If there is no null terminator, probably segfault!
  - **Constant strings** - strings in quotes.
    * Stored in **static memory**.
    * **Read only** global variables!

```
char *foo = "this is a constant";
char *boo = "this is a constant"; // Same string in memory
foo[1] = 0;
  // Immediate crash
  // To avoid this, write char foo[] = "this is a constant";
```

  - String-related functions/libraries:

```
// In string.h - string processing header:
char * strcpy(char *dest, const char *src);
  // String copy - must allocate space for dest
  // Or char * strncpy(char *dest, const char *src, size_t num);
    // Limits length copied; does not copy null terminator if too short
  // Could use void * memcpy(void *dest, const void *src, size_t n);
int strcmp(const char *str1, const char *str2);
  // String compare
char * strcat(char *dest, const char *src);
  // String concatenation
  // Or char *strncat(char *dest, const char *src, size_t n);
size_t strlen(const char *str);
  // String length - does not include '\0'

// In stdio.h - standard library for input/output:
// stdin and stdout functions:
char getchar();  // Reads in a char from stdin
char * gets();   // Reads in a string from stdin
                 // Terminates when '\n' or EOF; prone to buffer overflow
                 // Removed in C11
int printf(const char *format, ...); // Formatted output to stdout
int scanf(const char *format, ...); // Formatted input from stdin
```

```
                                    // Take pointers/address as arguments
                                    // int i; scanf("%d", &i);
// File input and output functions:
char fgetc(FILE *stream);
  // File getchar
  // Or char getc(FILE *stream);
char * fgets(char *buffer, int count, FILE *stream);
  // File gets() with length limit - fills '\0' if too long.
int fprintf(FILE *stream, const char *format, ...);
  // Formatted output to file
int fscanf(FILE *stream, const char *format, ...);
  // Formatted input from file
  // Take pointers/address as arguments
```

– If you use *gets*(), **buffer overflow** may be exploited.

Arguments in *main()*

To get arguments to the main function,

```
int main(int argc, char *argv[])
  // argc is the number of strings on command line, executable included
  //   In $ sort myFile 2, argc == 3
  // argv is an array of pointers to strings, executable included
  //   In $ sort myFile 2, argv[0] = "sort", argv[1] = "myFile", argv[2] =
      "2"
```

Endianness

Consider the following:

```
union confuzzle { int a; char b[4]; };
union confuzzle foo;
foo.a = 0x12345678;
```

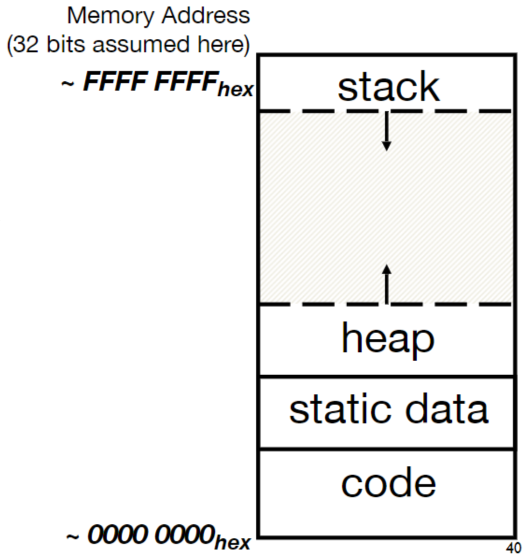
In a 32b architecture, foo.b[0] could be 0x12 or 0x78 depending on the architecture's **endianness**:

- **Big endian** - the first character is the most significant byte - 0x12;
- **Little endian** - the first character is the least significant byte - 0x78.

To handle this between different architecture, we have endian conversion functions ntohs(), htons(), ntohl(), htonl().

C Memory Management

Now, assuming that one program runs at a time with access to the entire memory, program's address space contains 4 regions:



- **Stack** - grows downward; *automatic* last in, first out (LIFO) data structure
  - When a function is called, allocate a new **"stack frame"** and have the **stack pointer** point to its start. Stack frames uses *contiguous blocks* of memory.
  - Included in which are:
    - Return address,
    - Arguments,
    - Space for **local variables**.
  - When the function ends, have the stack pointer move up, and free memory for future stack frames.
- **Heap** - grows upward; *manual* management with functions
  - **malloc()** - allocate a block of *uninitialized* memory.

```c
typedef struct { ... } TreeNode;
TreeNode *tp = (TreeNode *) malloc (sizeof(TreeNode));
```

  - **calloc()** - allocate a block of *zeroed* memory.
  - **free()** - free previously malloc-allocated block of memory.

```c
... // Continued
free((void *) tp); // Must be original address returned from malloc()
```

  - **realloc()** - change size of previously allocated block.
    After *realloc* the block might move; *realloc* **would NOT update other pointers pointing at the same block**.

```c
int *ip = (int *)malloc(sizeof(int)); // Always check for ip == NULL
ip = (int *)realloc(ip, 0); // Identical to free(ip)
```

- **Static data** - does not grow or shrink
  - **Global variables** outside functions.
  - Loaded when program starts, **can be modified**.
- **Code** - does not grow or shrink
  - Loaded when program starts, **does not change**.

Here, wrong manipulation of the heap memory is the biggest source of bugs in C code:

- Forget to deallocate memory - **memory leak!**

   - **Fragmentation!** - Less memory and slower precessing.

- Use data after calling free - **use after free!**

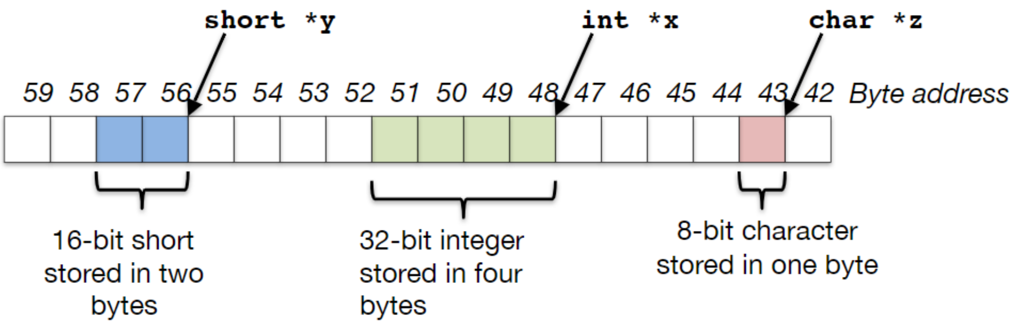
```c
// Exploiting Use-After-Free
name = (char *)malloc(sizeof(char) * 1024);
free(name);
foo = (char *)malloc(sizeof(char) * 1024);
  // Possibly foo and name have same address
scanf("%s\n", foo);
  // Attacker can write to foo whatever they want to name
if(name == "root") ...
```

- Free the same memory twice - **double free!**

```c
// Double-free can create use-after-free
name = (char *)malloc(sizeof(char) * 1024);
...
free(name);
...
foo = (char *)malloc(sizeof(char) * 1024);
...
free(name); // foo might get freed
```

Word Alignment

In C programming, oftentimes, we would prefer "word alignment", i.e. every piece of data starts on a 4 Byte boundary. In fact, some processors will not allow you to access 32b values without being on such a 4 Byte boundary; others will generally be slow if you are not on such a boundary.



The graph above represents the default alignment rules, assuming a 32b architecture:

- char: 1 byte - no alignment needed
- short: 2 bytes - half-word aligned
- int: 4 byte - word aligned

Therefore, on a 32b architecture, struct *foo*

```c
typedef struct foo{
        int a;
        char b;
        foo *c;
}foo;
```

will look like:

- 4b for a;
- 1b for b;
- 3b unused;
- and 4b for c.

And hence we have *sizeof(struct foo) == 12*. To save memory space, you should put all non-self-aligning structures together, however the architecture changes.

To dynamically allocate a 10 entry array of foos:

```
foo *f = (foo *)malloc(sizeof(foo) * 10);
  // Entries can be accessed as in an array:
  //    f[4].b = ‘\n’;
```

4 Floating-Point Arithmetic

Previously, we have introduced multiple ways to represent integers. Now we want to represent real numbers using a similar amount of bits; in fact, we want to represent both very large and very small numbers. We refer to the scientific notation in decimal and convert it to binary:

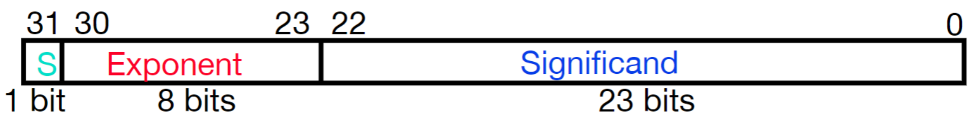
$$1.01_2 \times 2^{-1} = (1 \cdot 2^0 + 1 \cdot 2^{-2}) \cdot 2^{-1} = 0.625_{10}.$$

Computer arithmetic that supports it is called **floating point**, because it represents numbers where the binary point is not fixed, as it is for integers. They have analogous arithmetic as decimal numbers.

IEEE 754 Floating-Point Standard

Single precision floating points are represented in a 32b word, with:

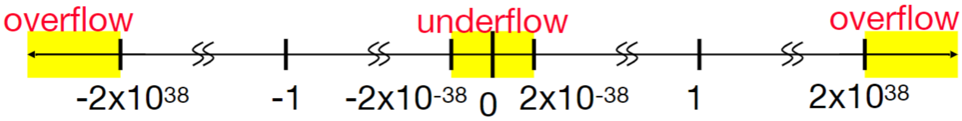
- 1 bit for **sign (s)** of floating point number,
  - 1 for negative, 0 for positive.
- 8 bits for **exponent (E)**,
  - **Biased notation** with bias -127.
- 23 bits for **fraction (F)** called **Significand**.
  - One extra bit of precision since **leading 1** is implicit.
  - Sign-magnitude (not 2’s complement).



This represents the number

$$(-1)^s \cdot (1 + F) \cdot 2^E.$$

Can represent approximately numbers in the range of $\pm 2.0 \cdot 10^{-38}$ to $\pm 2.0 \cdot 10^{38}$. If a number not in that range is casted onto floating point it will get an **overflow** or an **underflow**, and the programmer will be alerted.



Double Precision and C

To represent larger numbers, we have the IEEE 754 **Double Precision Standard** (64 bits):

- 1 bit for **sign (s)**,
- 11 bits for **exponent (E)**,
  - **Biased notation** with bias -1023.
- 52 bits for **fraction (F)**.

In C, such variables are declared as *double* - for single precision, declare as *float*. Apparently, you can save bits by modifying how you represent floating points.

Special cases in IEEE 754

To represent 0,

- Exponent all 0s; significand all 0s; either sign.
- Without denorms, it is just an approximation.

To represent $\pm\infty$,

- Exponent **all 1s**; significand **all 0s**.
- In floating point, division by 0 creates $\pm\infty$ for future computations with ∞.

To represent $\sqrt{-4.0}$ or 0/0, we create this concept of **NaN - Not a Number**:

- Exponent **all 1s**; significand **non 0**.
- Any operation with NaN results in NaN.
- Help with debugging - use significand to identify!

As you might have noticed, as we approach zero, the gap between the numbers is large. Hence we use **denormalized number**:

- Exponent **all 0s**; significand **non 0**.
- **No implied leading 0** and **implicit exponent = -126**.
- Smallest representable positive number 2^{-149}, second smallest 2^{-148}.

Sorting with Floating Point

- Sort sign field by just +/-.
- Sort exponent field (see as unsigned) by size.
- Sort significand (as it is sign-and-magnitude).
- Only need to consider NaN and $\pm\infty$!