CS2100 Comp Org Notes

AY23/24 Sem 1, github.com/gerteck

0. Computer Organisation

• Instruction set architecture (ISA) is the software stack and below it is the hardware stack.

• High level programming language → Assembly Language → Machine Code

• We first simplify the processor to three components: Arithmetic Logic Unit (ALU), Control Unit and Memory Unit.

1. C syntax

```
#include <iostream>
#include <stdio.h>
int main(void) {
    printf("Hello World!\n");
    std::cout << "Hello World!\n";
    return 0;
}
```

• C programs generally structured as such:

```
#preprocessor directives
main function header {
    declaration of variables
    executable statements
}
```

• **Preprocessor:** Starts with #, `#include` allows us to use codes defined in another file, `#define` allows us to define a constant.

• Always declare variables at the beginning of a function, and initialize (assign initial value) before use.

• Three common C preprocessors: Header file inclusion, Macro expansions, conditional compilations.

• C is **statically typed** language, type is the **property of the variable**. Once declared, variable can only store data of particular type.

Variable attributes: Name, Type, Address, Value.

• Names are case-sensitive, camelCase, PascalCase.

• All data in C is (or can be) represented as integers. Characters are represented by 8-bit "char" integers based on the ASCII table.

• Strings are then represented as: Array of char. Terminated by a null character (' \0' or 0)

Primitive Data Types in C:

Type	Size	Usage	Examples	Py	JS
int	4 bytes	Whole numbers	Max: 2^{31}-1, Min: -2^{31}	int	number
float	4 bytes	Real numbers	12.34f, 1.5e-2f [1]	float	number
double	8 bytes	Real numbers	12.34, 1.5e-2	float	number
char	1 byte	Characters	'A', 'z', '\n', ' ', '2'	str	string

Format Specifiers:

Placeholder	Type	Fn Use		Placeholder	Type	Fn Use
%c	char	printf/scanf		%lf	double	scanf
%d	int	printf/scanf		%p	pointers	printf
%f	float/double	printf/scanf				

Escape Sequences:

Sequence	Meaning	Result		Sequence	Meaning	Result
\n	New line	Ouput new line		%%	Percent	Display percent %
\t	Horizontal Tab	Move cursor		%p	pointers	printf
\"	Double quote	Display "		\a	alarm	Gen. bell sound

Typecasting in C:

```
/* syntax: (type) expression */
int ii = 5; float ff = 15.34
float a = (float) ii / 2; // a = 2.5
float b = (float) (ii / 2); // b = 2.0, floor division
int c = (int) ff / ii; // c = 3
```

Assignment Statements

• The value assigned is returned as result of evaluation.

```
a = b = c = 3 + 6;    // is possible
a = 5 + (b = 3);      // b = 3, a = 8
```

Associativity & Precedence

Operator Type	Operator	Associativity
Primary expression operators	() expr++ expr--	Left to right
Unary operators	* & + - ++expr --expr (typecast)	Right to left
Binary operators	* / %	Left to right
	+ -	
Assignment operators	= += -= *= /= %=	Right to left

(INCREASING PRECEDENCE)

Selection

• We may define our own boolean library or (`#include <stdbool.h>`).

Non-zero values treated as true, but only 1 (==) equal true.

```
#define false  0
#define true   1
#define bool   char
```

• Short-circuit evaluation.

switch/case:

• **fall through behavior**: Removal of `break` allows subsequent cases to run.

```
switch(<variable_or_expression>) {
  case value1:
    /* ... */
    break; // Prevents spill over to next case

  case value2:
    /* ... */
    // no break can spill over to next case

  case value3:
    /* ... */
    break;

  default: // code to execute if equal none.
    /* ... */
    break;
}
```

loops:

```
while ( condition )
{
    // loop body
}
```

```
do
{
    // loop body
} while ( condition );
```

```
for ( initialization; condition; update )
{
    // loop body
}
```

Initialization: initialize the **loop variable**

Condition: repeat loop while the condition on **loop variable** is true

Update: change value of **loop variable**

2. C syntax (Pointers & Functions)

Every **memory location** in a computer is indexed with an address.
All variables in C must be stored in memory,

```c
int main(void) {
    int a = 3, *b;       // b is a pointer to an int
    b = &a;              // b points to the address of a
    *b = 5;              // set a through b, a=5

    int *a_ptr;
    a_ptr = &a;
}
```

• **pointer variable** stores the address of another variable.

• **&** → address operator.
  `&x` → address of memory cell where value of x is stored, gets address of a variable.

• `*` → declares a pointer.
  `type *pointer_name` (e.g. `int *x`)

• `*` - dereferencing (access variable through pointer)
  `*x = 32` : following through the pointer to get the value

• Incrementing a pointer('s pointed value): `(*p)++;`
  without brackets: increments pointer to next address (depending on size of the data type) aka += sizeof(*p1)

```c
double a, *b;
b = &a; // legal
double c, d;
*d = &c; // legal
double e, f;
f = &e; // ILLEGAL!
```

Call-by-Value / Pointer

• In C, the actual parameters are passed to the formal parameters by a mechanism known as call-by-value.

• The only way for a function to modify the value of a variable outside its scope, is to use pointers to access that variable. (Call-by-pointer)

3. C Arrays, Strings & Structs

Arrays

• a homogenous collection of data all of the same type, occupying contiguous memory locations.

• declaration: `arr = elementType[size]`

• arr refers to `&arr[0]`

• an array name is a **fixed (constant) pointer**, which points to the first element in the array and cannot be reassigned - `arr1 = arr2` is illegal.

```c
// an array can ONLY be initialised at the time of declaration.
int evens[5] = {2, 4, 6, 8, 10};
// if you initialise values, no need to declare length
int odds[] = {1, 3, 5};
// uninitialised values will be zero value
int some[5] = {1, 2, 3}; // some = [1, 2, 3, 0, 0]

int numbers[3];
printf("Enter 3 integers:");
for (i = 0; i < 3; i++) {
 scanf("%d", &numbers[i]); }
```

In function prototypes

```c
// parameter names are optional
int sumArray(int [], int); // valid
int sumArray(int arr[], int size); // valid
int sumArray(int *, int); // pointer is valid too

// size can be specified but will be ignored
int sumArray(int arr[8], int size);

// function definition
int sumArray(int *arr, int size) { ... }
int sumArray(int arr[8], int size) { ... } // size ignored
```

Strings

• array of characters terminated with a null character: \ 0, which has ASCII value of 0.

• string functions: `#include` <string.h>

```c
char my_str[] = "hello";
char my_str[] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

I/O

• `in` : `fgets(str, size, stdin)` reads (size - 1) chars or until newline encountered.

• `in` : `scanf("%s", str)` - reads until whitespace.

• `out` : `puts(str)` - terminates with newline.

• `out` : `printf("%sn", str)` - prints until '\0' in `str` encountered.

String functions

• `strlen(s)` : returns number of characters in s up to '\0'

• `strcmp(s1, s2)` : compares the ASCII values of corresponding characters, returns s1 - s2, negative number / positive number / 0 if they are equal.

• `strncmp(s1, s2, n)` : strcmp for first n characters of s1 and s2

• `strcpy(s1, s2)` : copy s2 into s1, ○cannot directly assign s1 = "Hello", but can copy: strcpy(s1, "Hello")

• `strncpy(s1, s2, n)` : copy first n characters of s2 into s1

Structs

• allow grouping of heterogenous data

• passed by value into functions
  unless: passing array of structs to a function
  array members of structs are **deeply copied**

• can be reassigned

• no memory is allocated to a type.

create new types called box_t and nested_box_t:

```c
// declare BEFORE function prototypes
typedef struct {
 int length, width;
 float height;
} box_t;

typedef struct {
 int id;
 box_t smaller_box;
} nested_box_t;

// initialising struct variables
box_t mybox = {2, 3, 5.1};
nested_box_t big_box = {0, {4, 3, 6.7}};

// accessing members
box.length = 1;
big_box.smaller_box.width = 2;
```

Arrow Operator −>

• `(*player_ptr).name` is equivalent to `player_ptr−>name`

• `*player_ptr.name` means `*(player_ptr.name)` (dot has higher precedence

4. Number Systems

Data Representation

- 1 byte = 8 bits

- word = multiple of a byte (e.g. 1 byte, 2 bytes, 4 bytes)
  64-bit machine → 1 word is 8 bytes

- N bits can represent up 2^N to values

- to represent values: ceil $[log_2 M]$ bits required

Weighted Number systems

- weighted number system → has a base (radix)
  base/radix R has weights in powers of R

Prefixes in C

- prefix `0` for octal (e.g. `032` = $(32)_8$)

- prefix `0x` for hexadecimal (e.g. `0x32` = $(32)_16$)

- prefix `0b` for binary

Conversion

- **decimal to binary:**
  whole numbers: repeated **division** by 2, LSB → MSB
  fractions: repeated **multiplication** by 2, MSB → LSB

- **decimal to base-R:**
  for whole numbers: repeated **division** by R for fractions:
  repeated **multiplication** by R

- binary → octal: partition in groups of 3

- octal → binary: convert each digit into 3-bit binary

- binary → hexadecimal: partition in groups of 4

- hexadecimal → binary: convert each digit to 4-bit binary

ASCII

- American Standard Code for Information Interchange

- 7 bits plus 1 parity bit (for error checking): $2^7 = 128$

- in C: `char` datatype is 1 byte = 8 bit integer
  corresponds to ASCII - can typecast int/char
  e.g. convert uppercase char to lowercase: `c = c + 'a'− 'A'`

Negative Numbers

- **unsigned** numbers: only non-negative values
- **signed** numbers: include all values (positive and negative)
- for negating non-whole numbers: same as whole numbers
  (ignore the decimal point, then put it back)

Overflow

- positive + positive = negative, OR
- negative + negative = positive

1s addition: If carry out, add 1 to the result (wrap around)

```
  -2      1101            -3      1100
+ -5    + 1010          + -7    + 1000
----    -------         ----    -------
  -7     10111           -10     10100
----    +    1           ----    +    1
        -------                  -------
         1000                     0101
        -------                  -------
```
No overflow   Overflow!

2s addition:: Ignore the carry out.

Sign-and-Magnitude:

- MSB represents the sign (0 is positive)
- **range (8-bit):** -127_{10} to $+127_{10}$
  2 zeroes: `00000000` $(+0_{10})$ and `10000000` (-0_{10})
- **negating a number**: reverse the first bit
- **issues**
  1. there are two zeroes (which may be useful for limits!)
  2. not good for performing arithmetic due to the zero in
  front

1s Complement:

- negated value of x, $-x = 2^n - x - 1$
- **negating a number**: invert the bits
- **range (8-bit)**: -127_{10} to $+127_{10}$
  2 zeroes: `00000000` $(+0_{10})$ and `11111111` (-0_{10})
  range (n-bits): -2^{n-1} to 2^{n-1}

2s Complement

- = 1s complement + 1
- negated value of x, $-x = 2^n - x$
- **negating a number**
  invert the bits, then **add 1**
- **range (8-bit):** -128_{10} to 127_{10}
  zero: `00000000` $= +0_{10}$ **range (n-bits):** -2^{n-1} to $2^{n-1}-1$
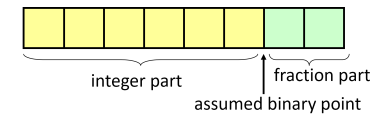
Excess Representation

- Allows the range of values to be distributed evenly between
  the positive and negative values, by a simple translation.
- `00..00` $= -2^n$
- `10..00` $= 0$
- to express n in Excess-M representation: $n + M$
  E.g. express 5 in excess 8 (4 bit): $5 + 8 = 13$ OR `1101`

5. Number Representations

Fixed-point representation

- In fixed-point representation, the number of bits allocated
  for the whole number part and fractional part are fixed.
- Issue: limited range.

integer part   fraction part
          assumed binary point

- If 2s complement is used, we can represent values like:
  $011010.11_{2s} = 26.75_{10}$
  $111110.11_{2s} = -000001.01_2 = -1.25_{10}$

Floating-point representation

- IEEE 754 floating-point representation
  ○ exponent is **excess-127**

- 3 components: **sign**, **exponent** and **mantissa (fraction)**

sign	exponent	mantissa

single-precision (32 bit format): 1-bit sign / 8-bit exponent
/ 23-bit mantissa

- **mantissa** is normalised with an implicit leading bit 1
  to maximise the numbers to be stored
  normalise it to the rightmost bit is always 1, no need to
  store it.

- better range and accuracy, but more complex

6. **MIPS + ISA**