C-Refresher: Session 03
Data Representation

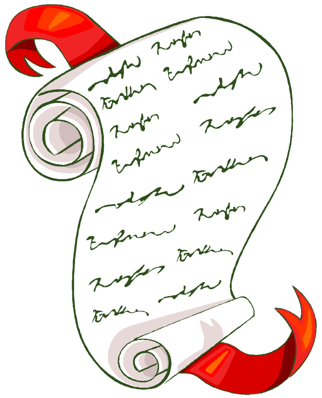
Arif Butt
Summer 2017

I am Thankful to my student Muhammad Zubair bcsf14m029@pucit.edu.pk for preparation of these slides in accordance with my video lectures at

http://www.arifbutt.me/category/c-behind-the-curtain/

Today's Agenda

- Data Types
- Multi-Byte Load/Store
- Fixed Point Representation
- IEEE Standard for Floating Point
- Range on Single Precision
- Precision



Data Types

A datatype, in programming, is a classification that specifies which type of value a variable can store and what type of mathematical, relational or logical operations can be applied to it without causing an error.

A `string`, for example, is a datatype that is used to classify text, and an `int` is a datatype used to classify whole numbers.

Data Types(cont…)

- Different datatypes are available in C for storing a particular type of values
- There are three types of values
  1. Integer
  2. Character
  3. Floating Point
- Different datatypes for storing a particular type of values are shown on next slide

Different Data Types

Integer	**Character**	**Floating Point**
`short`	`char`	`float`
`int`		`double`
`long`		`long double`
`long long`		

Note: `short`, `int`, `long`, `long long` and `char` are both signed and unsigned

Data Types(cont…)

❑Range:

- Range of values that can be occupied by different datatypes depends upon the platform, hardware (OS 32 or 64-bit) and compiler

- The command used to measure size of different datatypes is

```
sizeof(data_type);
```

Data Types(cont…)

❑limits.h

- There is a file `limits.h` which contains ranges for different datatypes
- Path of file is
  - `/usr/include/limits.h`

❑getconf

- Instead of looking at `limits.h` file, we can use `getconf` command which contains ranges of lots of parameters

```
$ getconf -a
```

Data Types(cont…)

- `getconf` command can also be passed an argument to show the value of that particular argument
- e.g:

```
$ getconf CHAR_MIN        //-128
$ getconf CHAR_MAX        //127
$ getconf UCHAR_MAX       //255
```

Data Types(cont…)

```
//Program showing sizes of different data types
#include<stdio.h>
int main(){
     printf("size of char: %d\n",sizeof(char));
     printf("size of short: %d\n",sizeof(short));
     printf("size of int: %d\n",sizeof(int));
     printf("size of long: %d\n",sizeof(long));
     printf("size of long long: %d\n",sizeof(long long));
     printf("size of float: %d\n",sizeof(float));
     printf("size of double: %d\n",sizeof(double));
     printf("size of long double: %d\n",sizeof(long double));
     return 0;}
```

Data Types(cont…)

❑output of above program:

- `size of char: 1`
- `size of short: 2`
- `size of int: 4`
- `size of long: 8`
- `size of long long: 8`
- `size of float: 4`
- `size of double: 8`
- `size of long double: 16`
- Note: These are the sizes on a `x86_64` system with kernel `4.6.0-kali-amd64`

Multi-Byte Load/Store

- Let's declare a variable

```
short i=54;
```

- $$54_{(10)} = \underbrace{0000\ 0000}_{\text{Byte 2}}\ \underbrace{0011\ 0110}_{\text{Byte 1}}{}_{(2)}$$
- Now there are more than one bytes
- There are two ways of storing these bytes in the memory
  - Little Endian scheme (used in intel)
  - Big Endian scheme(used in MIPS)

Multi-Byte Load/Store(cont…)

❑Little Endian:

- In Little Endian scheme, the bytes are put into the memory form right to left, i.e. the rightmost byte is put on a **lower** memory address and then the bytes from right to left are put in memory on consecutively **higher** memory addresses
- e.g.
- If we have memory addresses 100 and 101 then Byte-1 will be put in 100 memory address and Byte-2 will be put in 101

Multi-Byte Load/Store(cont…)

❑Big Endian:

- In Big Endian scheme, the bytes are put into the memory form left to right, i.e. the leftmost byte is put on a **lower** memory address and then the bytes from left to right are put in memory on consecutively **higher** memory addresses
- e.g.
- If we have memory addresses 100 and 101 then Byte-1 will be put in 101 memory address and Byte-2 will be put in 100

Multi-Byte Load/Store(cont…)

- Max number of values that can be stored using `n` number of bits can be calculated using the formula
  - 2^n
  - e.g.
  - No. of values stored in `1 bit` are 2^1,i.e. `1&0`
  - No. of values stored in `2 bits` are 2^2,i.e. `00, 01, 10, 11`
  - and so on
- Range of values that can be stored in `n` number of bits is given as(on next slide)

Multi-Byte Load/Store(cont…)

- **For Unsigned(n bits)**
  - $0 \rightarrow 2^n-1$
  - e.g. for 8-bits => $0 \rightarrow 2^8-1$ i.e. $0 \rightarrow 255$
- **For Signed(n bits)**

- There are two ways:

1. Signed Magnitude:

  - $-(2^{n-1}-1) \rightarrow +(2^{n-1}-1)$

- This way is generally not used in our computer systems due to two reasons

Multi-Byte Load/Store(cont…)

(i) Zero can be represented in two ways, i.e. we have a `+ve` zero `0000` and a `-ve` zero `1000` (as `0` represents a `+ve` sign and `1` represents `-ve` sign)

(ii) Normal Binary arithmetic rules do not apply

- e.g. adding `0001(+1)` and `1001(-1)` yields `1010(-2)`, it would rather have been 0 but its not

2. 2's Complement:

- $-2^{n-1} \rightarrow +(2^{n-1}-1)$
- e.g. for 8-bits => `-128 -> +127`

Multi-Byte Load/Store(cont…)

- 2's complement is used in computer systems as
  - zero can be represented in one way only, i.e. `0000` (if in 4-bits)
  - Binary arithmetic can be applied without any error
    - e.g. adding `0001(+1)` and `1111(-1)` yields `0000(0)`
- Note: There is an extra `-ve` number in 2's complement as there is only one way for representing zero

Multi-Byte Load/Store(cont…)

```
/*Program for getting range(s) of short datatype..may also
be used for some other*/
#include<stdio.h>
int main(){
  printf("Size of short: %d\n",sizeof(short));
  int bits=8*sizeof(short);
  printf("Bits: %d\n",bits);
  int from=0;
  int to=(1<<bits)-1;    //1*2^bits
  printf("Range of unsigned short is from %d to %d\n",from,to);
  from=-(1<<bits-1);
  to=(1<<bits-1)-1;
  printf("Range of short is from %d to %d\n",from,to);
  return 0;}
```

Multi-Byte Load/Store(cont…)

- Output of above program:

```
Size of short: 2
Bits: 16
Range of unsigned short is from 0 to 65535
Range of short is from -32768 to 32767
```

- Similarly, we can find range for other data types using this program as a template, i.e. replacing `short` with that datatype e.g. `int`
- These values can also be verified from `/usr/include/limits.h` file or using `getconf` command

Fixed Point Representation

- Real number can be represented in two ways
  - Fixed point
  - Floating point (our system uses this one)

❑**Fixed Point Representation:**

- Let's take a number $(12.6)_{10} = (1100.10011001\ldots)_2$
- There are three fields in fixed point representation
  - Sign (+, -)
  - Integer field
  - Fractional field

Fixed Point Representation(cont…)

- If we represent the number in 32-bit system

1-bit	15-bits	16-bits
0	000000000001100	1001100110011001
Sign (0/1)	Integer part	Fractional part

- Now the largest number which can be stored is given as
  - $(2^{15}-1)+(1-2^{-16}) = 32767.9999 \approx 32768$
- Smallest number is
  - $0+2^{-16} \approx 0.000015$

Fixed Point Representation(cont…)

- **Advantages:**
  - Very fast performance as number is saved as integer
  - Perform different optimizing techniques without any additional hardware
- **Disadvantages:**
  - Operand size -- has very limited range of operand values

Floating Point Representation

- Introduced in 1985, based on scientific notation
- It has been accepted as the IEEE standard for floating point
- Current version of IEEE is **IEEE 754-2008**
- **Storage:**

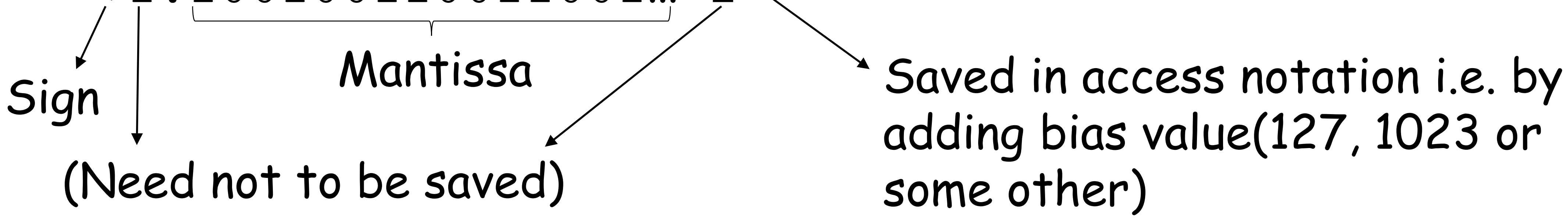
	Sign	Exponent	Mantissa
• Single precision of 32-bits	1-bit	8-bits	23-bits
• Double precision of 64-bits	1-bit	11-bits	52-bits
• Quadruple precision of 128-bits	1-bit	15-bits	118-bits
• Octuplet precision of 256-bits	1-bit	19-bits	236-bits

Floating Point Representation(cont…)

- **Sign field** can be `0` or `1` i.e. + or –
- In **Exponent field**, base is implicit i.e. the base is `2`
- The exponent can be both `+ve` and `-ve`
- To store these `+ve` and `-ve` exponents, a bias is added to the exponent, e.g.
  - In case of single precision, bias value is `127`
  - In case of double precision, bias is `1023`
  - e.g. in single precision
    - To store an exp. of `+3`, you actually store `127+3=130`
    - To store an exp. of `-3`, you actually store `127-3=124`

Floating Point Representation(cont…)

- Larger the number of bits for Exponent, the larger is the range
- Larger the number of bits for **Mantissa field**, the greater is the precision
- Let’s take an example of how a number is stored in floating point representation
  - $12.6_{10} = 1100.100110011001\ldots_2$
  - $+1.\underbrace{100100110011001\ldots}_{\text{Mantissa}} * 2^{\textcircled{+3}}$



Sign

Mantissa

(Need not to be saved)

Saved in access notation i.e. by adding bias value(127, 1023 or some other)

Floating Point Representation(cont…)

- So in single precision the above values will be stored in memory like

1-bit	8-bits	23-bits
0	1000 0010	10011001100110011001…
Sign	+3+127=130	Mantissa

Range on Single Precision

- **Smallest Value:**

1-bit	8-bits	23-bits
0/1	0000 0001	00000000000000000…
Sign	1-127=126	Mantissa

$\pm 1.0*2-126=\pm 2-126$

- **Largest Value:**

1-bit	8-bits	23-bits
0/1	1111 1110	11111111111111111…
Sign	254-127=+127	Mantissa

$\pm 1.1111*2^{+127}=\pm 2*2^{+127}$

Note: Exponents of all 0's and all 1's are reserved

Precision

- **floats:**
- `float` is stored in single precision which has 23-bits for decimal part
- $23*\log_{10}{}^{2} = 23*0.3 \approx 6$ (6 decimal digits per precision)
- **doubles:**
- `double` is stored in double precision which has 52-bits for decimal part
- $52*\log_{10}{}^{2} = 52*0.3 \approx 12$ (12 decimal digits per precision)

Overflow & Underflow

❑**Overflow:**

- A value larger than the largest magnitude value
- e.g. in single precision
- value$> 1.1111*2^{+127} = \infty$

❑**Underflow:**

- A value smaller than the smallest magnitude value
- e.g. in single precision
- value$<1*2^{-149} = 0$
- It may not have a very large effect on addition but have a very large effect on multiplication

Overflow & Underflow(cont…)

- There is a bunch of numbers which, along floating point numbers, get very small by sacrificing the significant bits, these numbers are called De-normalized numbers
- Numbers $<1*2^{-149}$ are de-normalized

Overflow & Underflow(cont…)

```
//Program for showing overflow
#include<stdio.h>
int main(){
    short a,b;
    printf("Enter a number: ");
    scanf("%d",&a);
    b=a+10;
    printf("%d+10=%d\n",a,b);
    return 0;
}
```

Overflow & Underflow(cont…)

- Output of above program is:

```
Enter a number: 32767
32767+10 = -32759
```

- Here, when we add $7FFF_{16}$(32767_{10}) and A_{16}(10_{10}), the result is 8009_{16}(-32759_{10})
- Actually 8009_{16}=$1000\ 0000\ 0000\ 1001_2$ (a –ve number)
- So after taking 2’s complement, we get -32759_{10}

SUMMARY