CS1100 – Introduction to Programming

Lecture 5: Revision of Main Ideas

Instructor: Shweta Agrawal (shweta.a@cse.iitm.ac.in)

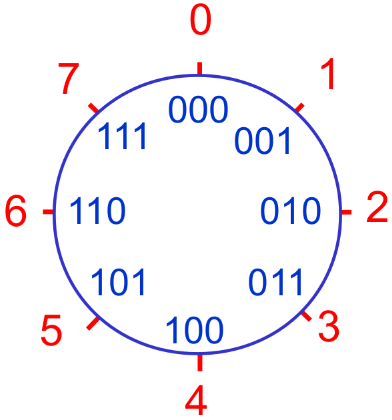
Representing values in Binary

If we have m bits, we can represent 2^m unique different values.

Representing values in Binary

If we have m bits, we can represent 2^m unique different values.

A useful circle :



Representing negative numbers

Sign Magnitude notation

- Use one bit for sign, others for magnitude of the number.

Representing negative numbers

Sign Magnitude notation

- Use one bit for sign, others for magnitude of the number.

			Sign Magn.
0	0	0	0
0	0	1	+1
0	1	0	+2
0	1	1	+3
1	0	0	0
1	0	1	-1
1	1	0	-2
1	1	1	-3

Representing negative numbers

Sign Magnitude notation

- Use one bit for sign, others for magnitude of the number.

			Sign Magn.
0	0	0	0
0	0	1	+1
0	1	0	+2
0	1	1	+3
1	0	0	0
1	0	1	-1
1	1	0	-2
1	1	1	-3

- using n bits: $-(2^{n-1}-1)\ldots(2^{n-1}-1)$.
- zero has two representations.

Representing negative numbers

Ones complement notation

- for a negative number n, represent the number by the bit complement of its binary representation.

Representing negative numbers

Ones complement notation

- for a negative number *n*, represent the number by the bit complement of its binary representation.

			Sign Magn.	Ones comp.
0	0	0	0	0
0	0	1	+1	+1
0	1	0	+2	+2
0	1	1	+3	+3
1	0	0	0	-3
1	0	1	-1	-2
1	1	0	-2	-1
1	1	1	-3	0

Representing negative numbers

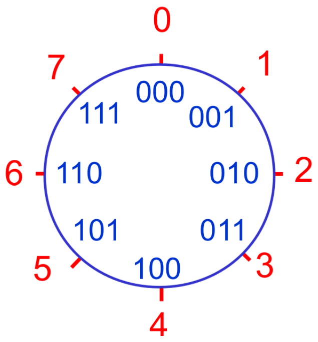
Ones complement notation

- for a negative number n, represent the number by the bit complement of its binary representation.

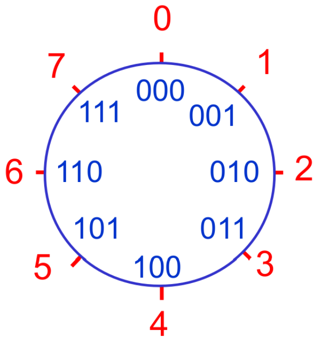
			Sign Magn.	Ones comp.
0	0	0	0	0
0	0	1	+1	+1
0	1	0	+2	+2
0	1	1	+3	+3
1	0	0	0	-3
1	0	1	-1	-2
1	1	0	-2	-1
1	1	1	-3	0

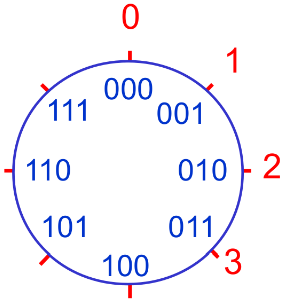
- zero has two representations.
- not very widely used representation.

Representing negative numbers - Twos complement

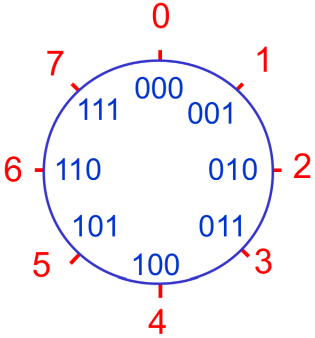


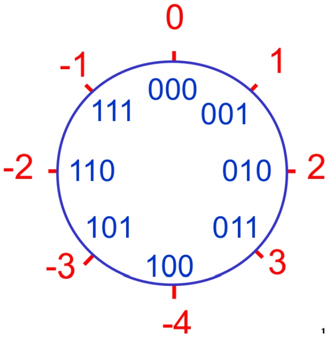
Representing negative numbers - Twos complement





Representing negative numbers - Twos complement





Representing negative numbers - Twos complement

- for a negative number $-n$, compute the number $2^k - n$, where k is the number of bits used to represent the value of n. The bit that represents the sign is extra.
- Two's complement for $-n$ has first bit 1 (representing minus) and remaining k bits encoding value $2^k - n$.

Representing negative numbers - Twos complement

- for a negative number $-n$, compute the number $2^k - n$, where k is the number of bits used to represent the value of n. The bit that represents the sign is extra.
- Two's complement for $-n$ has first bit 1 (representing minus) and remaining k bits encoding value $2^k - n$.

			Sign Magn.	Ones comp.	Twos comp.
0	0	0	0	0	0
0	0	1	+1	+1	+1
0	1	0	+2	+2	+2
0	1	1	+3	+3	+3
1	0	0	0	-3	-4
1	0	1	-1	-2	-3
1	1	0	-2	-1	-2
1	1	1	-3	0	-1

Representing negative numbers - Twos complement

- for a negative number $-n$, compute the number $2^k - n$, where k is the number of bits used to represent the value of n. The bit that represents the sign is extra.
- Two's complement for $-n$ has first bit 1 (representing minus) and remaining k bits encoding value $2^k - n$.

			Sign Magn.	Ones comp.	Twos comp.
0	0	0	0	0	0
0	0	1	+1	+1	+1
0	1	0	+2	+2	+2
0	1	1	+3	+3	+3
1	0	0	0	-3	-4
1	0	1	-1	-2	-3
1	1	0	-2	-1	-2
1	1	1	-3	0	-1

- widely used representation.

Representing negative numbers

Arithmetic with these representations

			Sign Magn.	Ones comp.	Twos comp.
0	0	0	0	0	0
0	0	1	+1	+1	+1
0	1	0	+2	+2	+2
0	1	1	+3	+3	+3
1	0	0	0	-3	-4
1	0	1	-1	-2	-3
1	1	0	-2	-1	-2
1	1	1	-3	0	-1

Representing negative numbers

Arithmetic with these representations

			Sign Magn.	Ones comp.	Twos comp.
0	0	0	0	0	0
0	0	1	+1	+1	+1
0	1	0	+2	+2	+2
0	1	1	+3	+3	+3
1	0	0	0	-3	-4
1	0	1	-1	-2	-3
1	1	0	-2	-1	-2
1	1	1	-3	0	-1

- $2 + (-3)$

Representing negative numbers

Arithmetic with these representations

			Sign Magn.	Ones comp.	Twos comp.
0	0	0	0	0	0
0	0	1	+1	+1	+1
0	1	0	+2	+2	+2
0	1	1	+3	+3	+3
1	0	0	0	-3	-4
1	0	1	-1	-2	-3
1	1	0	-2	-1	-2
1	1	1	-3	0	-1

- $2 + (-3)$
- $3 + (-2)$

More examples : The case of 4 bits

	corresp. dec. oper.
0011	+3
+0100	+ +4
0111 = +7	+7

correct result

Example (c)

carries: 1 1 1	corresp. dec. oper.
1110	−2
+1010	+ −6
11000 = −8	−8

correct result

Example (d)

Some Programs: Sum of 2 numbers

```
#include <stdio.h>

/* sum 2 integers */

int main() {
    int x = 98;
    int y = 99;
    int z;

    z = x+y;
    printf("%d\n", z);
    return 0;
}
```

- int : defines that x, y, z are of type integers.
- z = x+y : evaluates x+y and stores it in z.
- What will be output if we print z?

Basic operators in C

- Arithmetic operators: +, -, *, /

Basic operators in C

- Arithmetic operators: $+$, -, *, $/$
- Modulus operator: %
  x % y : remainder when x is divided by y.

Basic operators in C

- Arithmetic operators: $+$, -, *, $/$
- Modulus operator: %
  x % y : remainder when x is divided by y.
- Assignment operator: $=$

Basic operators in C

- Arithmetic operators: $+$, `-`, `*`, $/$
- Modulus operator: `%`
  x `%` y : remainder when x is divided by y.
- Assignment operator: $=$

Input statement: scanf

scanf(format-string, &var1, &var2, ... , &var3);

- scanf is a function which allows us to accept inputs.
- Usually functions take fixed number of parameters/ arguments.
- scanf takes variable number of arguments.
- Notice the **&** preceeding the variables.

Weighted sum of 2 numbers

- Recall x denotes marks in Maths, y denotes marks in Physics.
- We wish to calculate weighted total such that Maths marks are given 30% weightage and Physics marks are given 70% weightage.
- $z = \frac{30}{100}x + \frac{70}{100}y$.

Weighted sum of 2 numbers

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    int total;

    total = (30/100)*mathMarks + (70/100)*phyMarks;
    printf("%d\n", total);
}
```

Weighted sum of 2 numbers

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    int total;

    total = (30/100)*mathMarks + (70/100)*phyMarks;
    printf("%d\n", total);
}
```

- What is the output of the program?

Weighted sum of 2 numbers

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    int total;

    total = (30/100)*mathMarks + (70/100)*phyMarks;
    printf("%d\n", total);
}
```

- What is the output of the program?
- Is the variable total still guaranteed to be an integer?

Weighted sum of 2 numbers

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    float total;        /* float variable */

    total = (30/100)*mathMarks + (70/100)*phyMarks;
    printf("%f\n", total); /* change here */
}
```

Weighted sum of 2 numbers

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    float total;        /* float variable */

    total = (30/100)*mathMarks + (70/100)*phyMarks;
    printf("%f\n", total); /* change here */
}
```

- What is the output of the program?

Weighted sum of 2 numbers

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    float total;        /* float variable */

    total = (30/100)*mathMarks + (70/100)*phyMarks;
    printf("%f\n", total); /* change here */
}
```

- What is the output of the program?
- $\frac{30}{100}$ and $\frac{70}{100}$ evaluate to 0 and therefore total is zero.

Weighted sum of 2 numbers – a correct program

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    float total;

    total = (30.0/100)*mathMarks + (70.0/100)*phyMarks;
    printf("%f\n", total);
}
```

Weighted sum of 2 numbers – a correct program

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    float total;

    total = (30.0/100)*mathMarks + (70.0/100)*phyMarks;
    printf("%f\n", total);
}
```

- What is the output of the program?

Weighted sum of 2 numbers – a correct program

```
#include <stdio.h>

/* weighted sum 2 integers */

main() {
    int mathMarks = 98;
    int phyMarks = 99;
    float total;

    total = (30.0/100)*mathMarks + (70.0/100)*phyMarks;
    printf("%f\n", total);
}
```

- What is the output of the program?

Basic operators in C

- Arithmetic operators: +, -, *, /

Basic operators in C

- Arithmetic operators: $+$, -, *, /
- Modulus operator: %
  x % y : remainder when x is divided by y.

Basic operators in C

- Arithmetic operators: +, -, *, /
- Modulus operator: %
  x % y : remainder when x is divided by y.
- Assignment operator: =

Basic operators in C

- Arithmetic operators: $+$, -, *, $/$
- Modulus operator: %
  x % y : remainder when x is divided by y.
- Assignment operator: $=$

Increment / decrement operators

- ++, - -
- prefix and post-fix only to a variable.

Increment / decrement operators

- ++, - -
- prefix and post-fix only to a variable.

```
#include<stdio.h>

int main() {
    int x, y;

    int n = 10;

    x = n++;
    y = ++n;
    printf(" x =  %d, y = %d\n", x, y);
    return 0;

}
```

Assignment operator =

Form: variable-name = expression

- z = x+y
- x+y = z   Incorrect form

Assignment operator =

Form: variable-name = expression

- z = x+y
- x+y = z   Incorrect form
- Assignment between different data types.
  - What happens if you assign float to int and vice versa?

Assignment operator =

Form: variable-name = expression

- z = x+y
- x+y = z   Incorrect form
- Assignment between different data types.
  - What happens if you assign float to int and vice versa?
- Multiple assignments.
  - x = y = z = (a + b);
  - evaluations happen right to left.

Assignment operator =

Form: variable-name = expression

- z = x+y
- x+y = z   Incorrect form
- Assignment between different data types.
  - What happens if you assign float to int and vice versa?
- Multiple assignments.
  - x = y = z = (a + b);
  - evaluations happen right to left.
- x = x + 10 can be written as x += 10;
- instead of +, we can also have -, *, /, %

Integers in C and Storage

- We have used **int** and **float** data types till now.

Integers in C and Storage

- We have used **int** and **float** data types till now.
- An integer variable is assigned 2 bytes (16 bits) to be stored. (Sometimes 4 bytes).

Integers in C and Storage

- We have used **int** and **float** data types till now.
- An integer variable is assigned 2 bytes (16 bits) to be stored. (Sometimes 4 bytes).
- In the 2s complement form this allows storage of values from

$$-2^{15} \text{ to } 2^{15} - 1$$

Integers in C and Storage

- We have used **int** and **float** data types till now.
- An integer variable is assigned 2 bytes (16 bits) to be stored. (Sometimes 4 bytes).
- In the 2s complement form this allows storage of values from

$$-2^{15} \text{ to } 2^{15} - 1$$

$$-32,768 \text{ to } 32,767$$

- There are limits to representation - we better choose the right type.

Integers in C and Storage

- We have used **int** and **float** data types till now.
- An integer variable is assigned 2 bytes (16 bits) to be stored. (Sometimes 4 bytes).
- In the 2s complement form this allows storage of values from

$$-2^{15} \text{ to } 2^{15} - 1$$

$$-32,768 \text{ to } 32,767$$

- There are limits to representation - we better choose the right type.
- What other data type can we use to store integers?

Integers in C and Storage

- We have used **int** and **float** data types till now.
- An integer variable is assigned 2 bytes (16 bits) to be stored. (Sometimes 4 bytes).
- In the 2s complement form this allows storage of values from

$$-2^{15} \text{ to } 2^{15} - 1$$

$$-32,768 \text{ to } 32,767$$

- There are limits to representation - we better choose the right type.
- What other data type can we use to store integers?
- **unsigned int**, **long**, **unsigned long**.

unsigned int

- Typically 4 bytes storage.
- Output an unsigned int: printf("%u", x);
- Input an unsigned int: scanf("%u", &x);
- Storage: binary format.

The Integers - The detailed Chart

int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

char

- Typically 1 byte storage.
- Every character has a unique code assigned to it (ASCII code).
  A = 65, B = 66

char

- Typically 1 byte storage.
- Every character has a unique code assigned to it (ASCII code).
  A = 65, B = 66
- Output a character: printf("%c", x);
- Input a character: scanf("%c", &x);

float

- Typically 4 bytes storage.
- Output a float: printf("%f ", x);
- Input a float: scanf("%f ", &x);
- How are fractions stored?

Binary vs decimal fractions

- $(10.11)_2 = (1 \times 2) + (0 \times 1) + (1 \times \frac{1}{2}) + (1 \times \frac{1}{2^2}) = (2.75)_{10}$

Binary vs decimal fractions

- $(10.11)_2 = (1 \times 2) + (0 \times 1) + (1 \times \frac{1}{2}) + (1 \times \frac{1}{2^2}) = (2.75)_{10}$
- $(0.90625)_{10} = (\quad)_2$
- $(0.9)_{10} = (\quad)_2$

Decimal Fraction → Binary Fraction (1)

Convert $(0.90625)_{10}$ to binary fraction

	0.90625	
	× 2	
	0.8125	+ integer part
1	× 2	
	0.625	+ integer part
1	× 2	
	0.25	+ integer part
1	× 2	
	0.5	+ integer part
0	× 2	
	0	+ integer part 1

$0.90625 = \frac{1}{2}(1+0.8125)$

$= \frac{1}{2}(1+ \frac{1}{2}(1+0.625))$

$= \frac{1}{2}(1+ \frac{1}{2}(1+ \frac{1}{2}(1+0.25)))$

$= \frac{1}{2}(1+\frac{1}{2}(1+ \frac{1}{2}(1+\frac{1}{2}(0+0.5))))$

$= \frac{1}{2}(1+\frac{1}{2}(1+\frac{1}{2}(1+\frac{1}{2}(0+\frac{1}{2}(1+0.0)))))$

$= ½+1/2^2+1/2^3+0/2^4 +1/2^5$

$= (0.11101)_2$

Thus, $(0.90625)_{10} = (0.11101)_2$

Decimal Fraction → Binary Fraction (2)

Convert $(0.9)_{10}$ to binary fraction

$$
\begin{array}{r l}
0.9 & \\
\times\, 2 & \\
\hline
0.8 & + \text{ integer part } 1 \\
\times\, 2 & \\
\hline
0.6 & + \text{ integer part } 1 \\
\times\, 2 & \\
\hline
0.2 & + \text{ integer part } 1 \\
\times\, 2 & \\
\hline
0.4 & + \text{ integer part } 0 \\
\times\, 2 & \\
\hline
0.8 & + \text{ integer part } 0
\end{array}
$$

For some fractions, we do not get 0.0 at any stage! These fractions require an infinite number of bits! Cannot be represented exactly!

Repetition

$$(0.9)_{10} = 0.11100110011001100\ldots = 0.1\overline{1100}$$

Binary vs decimal fractions

- $(10.11)_2 = (1 \times 2^1) + (0 \times 2^0) + (1 \times \frac{1}{2}) + (1 \times \frac{1}{2^2}) = (2.75)_{10}$
- $(0.90625)_{10} = (0.11101)_2$
- $(0.9)_{10} = (0.11100\textcolor{red}{11100}11100..)_2$

Fixed point vs floating point representation

Fixed point

- Position of radix point is fixed and is same for all numbers.
- Lets say we have 3 digits after radix point.

Fixed point vs floating point representation

Fixed point

- Position of radix point is fixed and is same for all numbers.
- Lets say we have 3 digits after radix point.
- $(0.120 \times 0.120)_{10} = (0.014)_{10}$
- A digit is lost.

Floating point

Fixed point vs floating point representation

Fixed point

- Position of radix point is fixed and is same for all numbers.
- Lets say we have 3 digits after radix point.
- $(0.120 \times 0.120)_{10} = (0.014)_{10}$
- A digit is lost.

Floating point

- $1.20 \times (10)^{-1} \times 1.20 \times (10)^{-1} = 1.44 \times (10)^{-2}$
- Wider range of numbers can be represented.
- IEEE standard: 32 bits are split as follows:
  - First bit for sign.
  - Next 8 bits for exponent.
  - Next 23 bits for mantissa.

Fixed point vs floating point representation

Fixed point

- Position of radix point is fixed and is same for all numbers.
- Lets say we have 3 digits after radix point.
- $(0.120 \times 0.120)_{10} = (0.014)_{10}$
- A digit is lost.

Floating point

- $1.20 \times (10)^{-1} \times 1.20 \times (10)^{-1} = 1.44 \times (10)^{-2}$
- Wider range of numbers can be represented.
- IEEE standard: 32 bits are split as follows:
  - First bit for sign.
  - Next 8 bits for exponent.
  - Next 23 bits for mantissa.
  - $(-39.9)_{10} = (-100111.11100)_2 = (-1.0011111100)_2 \times 2^5$.

Floats - different types

Type	Storage size	Value range
float	4 byte	1.2E-38 to 3.4E+38
double	8 byte	2.3E-308 to 1.7E+308
long double	10 byte	3.4E-4932 to 1.1E+4932

Output floats in C

printf(“ %w.p f ”, x);

- w.p is optional.
- w : total width of the field.
- p : precision (digits after decimal).

Output floats in C

printf(" %w.p f ", x);

- w.p is optional.
- w : total width of the field.
- p : precision (digits after decimal).

```
#include<stdio.h>
main() {

    float x = 2.00123;
    printf ("x = %5.4f\n", x);
    printf ("x = %8.7f\n", x);
}
```

Circumference of circle

```
#include<stdio.h>

main() {
    float radius;
    float circum;

    printf("Enter radius : ");
    scanf("%f", &radius);
    circum = 2* (22.0/7) * radius;

    printf ("radius = %f, circum = %f\n", radius, circum);
}
```

Circumference of circle

```
#include<stdio.h>

main() {
    float radius;
    float circum;

    printf("Enter radius : ");
    scanf("%f", &radius);
    circum = 2* (22.0/7) * radius;

    printf ("radius = %f, circum = %f\n", radius, circum);
}
```

- How to print output only upto 2 decimals?

Circumference of circle – formatted output

```
#include<stdio.h>

main() {
    float radius;
    float circum;

    printf("Enter radius : ");
    scanf("%f", &radius);
    circum = 2* (22.0/7) * radius;

    printf ("radius = %5.2f, circum = %5.2f\n", radius, ci
}
```

Output statement

`printf (format-string,` $var_1, var_2, ..., var_n$`)`

Output statement

`printf (format-string,` $var_1, var_2, ..., var_n$`)`

Format string specifies

- How many variables to expect?
- Type of each variable.
- How many columns to use for printing? (width)
- What is the precision? (if applicable)

Output statement

`printf (format-string,` $var_1, var_2, ..., var_n$`)`

Format string specifies

- How many variables to expect?
- Type of each variable.
- How many columns to use for printing? (width)
- What is the precision? (if applicable)
- Common mistakes:
  - mismatch in the actual number of variables given and those expected in the format string.

Formatted output

```
printf (``%w.pC", x);
```

Formatted output

```
printf (``%w.pC", x);
```

- w, p and C are place holders, can take different values.
- w: width of the output. (optional)
- p: precision of the output. (optional)
- C: Conversion character.

Formatted output

```
printf (``%w.pC", x);
```

- w, p and C are place holders, can take different values.
- w: width of the output. (optional)
- p: precision of the output. (optional)
- C: Conversion character.
  - d : integer
  - f : float
  - c : character
  - x : hexadecimal
  - o : octal
  - u : unsigned int
  - e : real decimal in exponent form

Input Statement

$$\texttt{scanf (format-string, } \&var_1, \&var_2, ..., \&var_n\texttt{)}$$

Input Statement

`scanf (format-string,` $\&var_1, \&var_2, ..., \&var_n$`)`

Format string specifies

- How many variables to expect?
- Type of each variable.

Input Statement

scanf (format-string, &var_1, &var_2, ..., &var_n)

Format string specifies

- How many variables to expect?
- Type of each variable.
- Common mistakes:
  - comma missing after the double quotes.
  - mismatch in the actual number of variables given and those expected in the format string.
  - & missing before the variable.