CTS8233F19

CST8233: Lab #2

IEEE Floating Point Representation

Objectives

The objective of this lab is to get the student familiar with the theory components covered in week 2.

Earning

There is no mark for this lab. However, each student should finish the lab's requirements within the lab session and demonstrate the working code to the instructor.

Laboratory Problem Description

Part A: The following C Program shows a code that uses casting to access and print the byte representations of different program data types.

Task A.1: Please copy and paste this code into your Visual Studio environment.
Task A.2: Run the program and notice how each data type is stored in the memory of your machine.
Task A.3: Change the value of each data type and notice how the stored bytes change according to the value.

```
/* $begin show-bytes */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

typedef unsigned char* byte_pointer;

void show_bytes(byte_pointer start, int len)
{
	int i;
	for (i = 0; i < len; i++)
		printf(" %.2x", start[i]);
	printf("\n");
}

void show_int(int x)
{
	show_bytes((byte_pointer)& x, sizeof(int));
}

void show_float(float x)
{
	show_bytes((byte_pointer)& x, sizeof(float));
}

void show_pointer(void* x)
{
	show_bytes((byte_pointer)& x, sizeof(void*));
}
/* $end show-bytes */
```

CTS8233F19

```c
/* $begin test-show-bytes */
void test_show_bytes(int val)
{
	int ival = val;
	float fval = (float)ival;
	int* pval = &ival;
	show_int(ival);
	show_float(fval);
	show_pointer(pval);
}
/* $end test-show-bytes */

void simple_show()
{
	/* $begin simple-show */
	int val = 0x12345678;
	byte_pointer valp = (byte_pointer)& val;
	show_bytes(valp, 1); /* A. */
	show_bytes(valp, 2); /* B. */
	show_bytes(valp, 3); /* C. */
	/* $end simple-show */
}

void float_eg()
{
	/* $begin float-show */
	int x = 543;
	float f = (float)x;
	show_int(x);
	show_float(f);
	/* $end float-show */
}

void string_eg()
{
	/* $begin show-string */
	char* s = "ABCDEF";
	show_bytes(s, strlen(s));
	/* $end show-string */
}

void show_twocomp()
{
	/* $begin show-twocomp */
	short int x = 12345;
	short int mx = -x;

	show_bytes((byte_pointer)& x, sizeof(short int));
	show_bytes((byte_pointer)& mx, sizeof(short int));
	/* $end show-twocomp */
}

int main(int argc, char* argv[])
{
	int val = 12345;

	if (argc > 1) {
		if (argv[1][0] == '0' && argv[1][1] == 'x')
			sscanf(argv[1] + 2, "%x", &val);
		else
```

```
                sscanf(argv[1], "%d", &val);
        printf("calling test_show_bytes\n");
        test_show_bytes(val);
    }
    else {
        printf("calling show_twocomp\n");
        show_twocomp();
        printf("Calling simple_show\n");
        simple_show();
        printf("Calling float_eg\n");
        float_eg();
        printf("Calling string_eg\n");
        string_eg();
    }
    return 0;
}
```

Part B: The following program demonstrate the overflow and underflow

Task B.1: Copy and paste the following code to your Visual Studio environment.
Task B.2: Run the code and notice the output of the program. Explain what happens to the instructor.
Task B.3: Change the loop's determinant "i" to few numbers higher than 127 and notice the output. Explain the results to the lab instructor.

```
/* $begin show-bytes */
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

#include<math.h>
int main()
{
    int i;
    float n, x;
    n = 1.0;
    for (i = 0; i <= 127; i++)
    {
        n = n * 2.0;
        x = 1.0 / n;
            printf("%d %e %E\n", i, x, n);
    }
    return 0;
}
```

Part C: In this task, you will evaluate the accuracy of Stirling's famous approximation:

$$n! \approx \sqrt{2 * \pi * n} * \left(\frac{n}{e}\right)^n$$

Write a program to output a table of the following form for n = 0 to 10:

n	$n!$	Stirling's	Absolute error	Relative error

Hint: If your computer system does not have a predefined value of π, then you can use either:

CTS8233F19

$$\boldsymbol{\pi = acos\,(-0.1)} \text{ OR } \boldsymbol{\pi = 4.0 * atan\,(1.0)}.$$

n	$n!$	Stirling's	Absolute error	Relative error
1	1	0.922137	0.077863	7.7863
2	2	1.919004	0.080996	4.0498
3	6	5.836210	0.163790	2.7298
4	24	23.506175	0.493825	2.0576
5	120	118.019168	1.980832	1.6507
6	720	710.078185	9.921815	1.3780
7	5040	4980.395832	59.604168	1.1826
8	40320	39902.395453	417.604547	1.0357
9	362880	359536.872842	3343.127158	0.9213
10	3628800	3598695.618741	30104.381259	0.8296

Your numbers may be slightly different depending on the computer system and the precision used. Judging from the results, does the accuracy increase or decrease with increasing n?