Week 03 Laboratory Sample Solutions

Objectives

- to practice using bitwise memory
- to practice manipulating dynamic memory
- to explore arbitrary precision integer arithmetic

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab03`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab03
$ cd lab03
$ 1521 fetch lab03
```

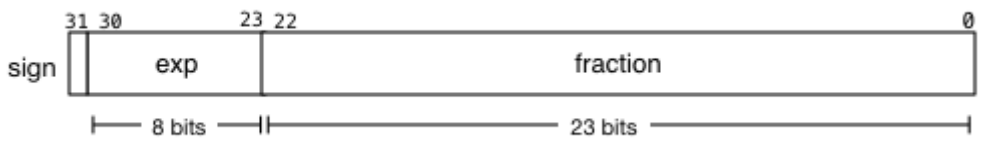
Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE — INDIVIDUAL:

Extract The Components of a Float

Floating-point Representation

Single-precision floating-point numbers that follow the IEEE 754 standard have an internal structure that looks like this:



The value is determined as:

$$-1^{\text{sign}} \times (1 + \text{frac}) \times 2^{\text{exp}-127}$$

The 32 bits in each `float` are used as follows:

`sign` is a single bit that indicates the number's sign. If set to 0, the number is positive; if set to 1, the number is negative.

`exp` is an unsigned 8-bit value (giving a range of $[0 \cdots 255]$) which is interpreted as a value in the range $[-127 \cdots 128]$ by subtracting 127 (the “bias”) from the stored 8-bit value. It gives a multiplier for the fraction part (i.e., $2^{\text{exp}-127}$).

`frac` is a value in the range $[0 \cdots 1]$, determined using positional notation:

$$\frac{\text{bit}_{22}}{2^1} + \frac{\text{bit}_{21}}{2^2} + \frac{\text{bit}_{20}}{2^3} + \cdots + \frac{\text{bit}_2}{2^{21}} + \frac{\text{bit}_1}{2^{22}} + \frac{\text{bit}_0}{2^{23}}$$

The overall value of the floating-point value is determined by adding 1 to the fraction: we assume that the "fraction" part is actually a value in the range $[1 \cdots 2]$, but save bits by not explicitly storing the leading 1 bit.

For example:

```
raw 32 bits:  01000000001000000000000000000000
partitioned:  0 10000000 01000000000000000000000
sign:         0, so positive
exp:          10000000 = 128, so multiplier is 2^(128-127) = 2^1
frac:         0×2^-1 + 1×2^-2 + ... = 0.25, but we need to add 1

final value:  1.25 × 2^1 = 2.5
```

Exercise Description

Your task is to add code to this function in **float_bits.c**:

```
// separate out the 3 components of a float
float_components_t float_bits(uint32_t f) {
    // PUT YOUR CODE HERE
}
```

The function `float_bits` is given the bits of a float as type `uint32_t`. Add code so that it returns a `struct float_components`, containing the sign, exponent and fraction fields of the `struct`.

You also should add appropriate code to the functions `is_nan`, `is_positive_infinity`, `is_negative_infinity`, and `is_zero`. All four functions take a `struct float_components` as their argument, and return 0 or 1 depending on some property of the `float` it represents:

- `is_nan` returns 1 iff the `struct float_components` corresponds to the not-a-number value, NaN, and 0 otherwise;
- `is_positive_infinity` returns 1 iff the `struct float_components` corresponds to the positive infinity, `inf`, and 0 otherwise;
- `is_negative_infinity` returns 1 iff the `struct float_components` corresponds to the negative infinity, `-inf`, and 0 otherwise; and
- `is_zero` returns 1 iff the `struct float_components` corresponds to either positive or negative zero.

These function must be implemented only using bit operations and integer comparisons.

Once these functions are completed, you should get output like:

```
$ ./float_bits -42
float_bits(-42) returned
sign=0x1
exponent=0x84
fraction=0x280000
is_nan returned 0
is_positive_infinity returned 0
is_negative_infinity returned 0
is_zero returned 0
$ ./float_bits 3.14159
float_bits(3.14159012) returned
sign=0x0
exponent=0x80
fraction=0x490fd0
is_nan returned 0
is_positive_infinity returned 0
is_negative_infinity returned 0
is_zero returned 0
$ ./float_bits -inf
float_bits(-inf) returned
sign=0x1
exponent=0xff
fraction=0x000000
is_nan returned 0
is_positive_infinity returned 0
is_negative_infinity returned 1
is_zero returned 0
```

Use *make* to build your code:

```
$ make    # or 'make float_bits'
```

HINT:

To extract the three components of the `float`, use the bitwise operators `&` and `>>`.

NOTE:

You may define and call your own functions if you wish.

You are not permitted to call any functions from the C library, or to perform floating-point operations, or to use variables of type `float` or `double`, or to use multiplication or division.

You are not permitted to change the `main` function you have been given, or to change the type `struct float_components` provided, or to change `float_bits`' prototype (its return type and argument types).

iff stands for "if and only if".

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest float_bits
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab03_float_bits float_bits.c
```

You must run `give` before **Monday 05 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `float_bits.c`

```c
// Sample solution for COMP1521 lab exercises
//
// Extract the 3 parts of a float using bit operations only

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

#include "floats.h"

#define SIGN_SHIFT       31
#define SIGN_MASK        0x1
#define EXPONENT_SHIFT   23
#define EXPONENT_MASK    0xFF
#define FRACTION_SHIFT   0
#define FRACTION_MASK    0x7FFFFF

#define EXPONENT_INF_NAN 0xFF

// separate out the 3 components of a float
float_components_t float_bits(uint32_t f) {
    float_components_t c;
    c.exponent = (f >> EXPONENT_SHIFT) & EXPONENT_MASK;
    c.fraction = (f >> FRACTION_SHIFT) & FRACTION_MASK;
    c.sign = (f >> SIGN_SHIFT) & SIGN_MASK;
    return c;
}

// given the 3 components of a float
// return 1 if it is NaN, 0 otherwise
int is_nan(float_components_t f) {
    return f.exponent == EXPONENT_INF_NAN && f.fraction != 0;
}

// given the 3 components of a float
// return 1 if it is inf, 0 otherwise
int is_positive_infinity(float_components_t f) {
    return f.exponent == EXPONENT_INF_NAN && f.fraction == 0 && f.sign == 0;
}

// given the 3 components of a float
// return 1 if it is -inf, 0 otherwise
int is_negative_infinity(float_components_t f) {
    return f.exponent == EXPONENT_INF_NAN && f.fraction == 0 && f.sign == 1;
}

// given the 3 components of a float
// return 1 if it is 0 or -0, 0 otherwise
int is_zero(float_components_t f) {
    return f.exponent == 0 && f.fraction == 0;
}
```

Alternative solution for `float_bits.c`

```c
// float_bits.c
// Sample solution for COMP1521 lab exercises
// Dylan Brotherston
//
// The following code changes main()
// in order to implement full error checking on the input
// when converting to a float
//
// Extract the 3 parts of a float using bit operations only

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

// Required for float conversion error checking
#include <errno.h> // errno
#include <math.h>  // HUGE_VALF

//
// We use this union to obtain the raw bits of a float
// by storing the float in the f field
// and then using the u field to obtain the bits
//
union overlay {
    float f;
    uint32_t u;
};

//
// A struct suitable for storing the 3 components of a float
//

typedef struct float_components {
    uint32_t sign;
    uint32_t exponent;
    uint32_t fraction;
} float_components_t;

float_components_t float_bits(uint32_t bits);
int is_nan(float_components_t f);
int is_positive_infinity(float_components_t f);
int is_negative_infinity(float_components_t f);
int is_zero(float_components_t f);

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        union overlay input;

        errno = 0; // error number: set to 0 to clear previous errors
        char *endptr =
            NULL; // create a pointer to pass to strtof for error checking
        input.f = strtof(argv[arg], &endptr); // strtof - string to float
        if (endptr == argv[arg]) {
            // If endptr is set to the start of the string
            // then no conversion occured, eg. invalid syntax
            fprintf(stderr, "%s: Could not convert \"%s\" to float\n", argv[0],
                    endptr);
            return 1;
        }
        if (*endptr != '\0') {
            // If endptr is not the start of the string AND not the end of the string
            // then a conversion occured but extra characters were appended
            fprintf(stderr,
                    "%s: Non digit characters after float value: \"%s\"\n",
                    argv[0], endptr);
            return 1;
        }
        if (errno) {
            // If errno is set the an error occored
            // only possible error is ERANGE - value out of range
            if (input.f == HUGE_VALF || input.f == -HUGE_VALF) {
                // input value was to large for a float
                // |value| > ~10^38
                fprintf(stderr
```

```
                "%s: Value to big, float will overflow: \"%s\"\n",
                argv[0], argv[arg]);
        }
        if (input.f == 0.0f) {
            // input value was to small for a float
            // |value| < ~10^-44
            fprintf(stderr,
                "%s: Value to small, float will underflow: \"%s\"\n",
                argv[0], argv[arg]);
        }
        return 1;
    }

    float_components_t c = float_bits(input.u);

    printf("float_bits(%.9g) returned\n", input.f);
    printf("sign = 0x%x\n", c.sign);
    printf("exponent = 0x%02x\n", c.exponent);
    printf("fraction = 0x%06x\n", c.fraction);

    printf("is_nan returned: %d\n", is_nan(c));
    printf("is_positive_infinity returned: %d\n", is_positive_infinity(c));
    printf("is_negative_infinity returned: %d\n", is_negative_infinity(c));
    printf("is_zero returned: %d\n", is_zero(c));
}
    }

    return 0;
}

// DO NOT CHANGE THE CODE ABOVE HERE

// separate out the 3 components of a float
float_components_t float_bits(uint32_t f) {
    return (float_components_t){ (f >> 31) & 0x1, (f >> 23) & 0xFF,
                                 (f >> 0) & 0x7FFFFF };
}

// given the 3 components of a float
// return 1 if it is NaN, 0 otherwise
int is_nan(float_components_t f) {
    return f.exponent == 0xFF && f.fraction > 0x0;
}

// given the 3 components of a float
// return 1 if it is inf, 0 otherwise
int is_positive_infinity(float_components_t f) {
    return f.sign == 0x0 && f.exponent == 0xFF && f.fraction == 0x0;
}

// given the 3 components of a float
// return 1 if it is -inf, 0 otherwise
int is_negative_infinity(float_components_t f) {
    return f.sign == 0x1 && f.exponent == 0xFF && f.fraction == 0x0;
}

// given the 3 components of a float
// return 1 if it is 0 or -0, 0 otherwise
int is_zero(float_components_t f) {
    return f.exponent == 0x0 && f.fraction == 0x0;
}
```

EXERCISE — INDIVIDUAL:

Multiply A Float by 2048 Using Bit Operations

Your task is to add code to this function in **float_2048.c**:

```c
// float_2048 is given the bits of a float f as a uint32_t
// it uses bit operations and + to calculate f * 2048
// and returns the bits of this value as a uint32_t
//
// if the result is too large to be represented as a float +inf or -inf is returned
//
// if f is +0, -0, +inf or -inf, or Nan it is returned unchanged
//
// float_2048 assumes f is not a denormal number
//
uint32_t float_2048(uint32_t f) {
    // PUT YOUR CODE HERE

    return 42;
}
```

Add code to the function `float_2048` so that, given a float as a `uint32_t`, it multiplies that value by 2048 using only bit operations and addition (+), and returns the result. If, after multiplication, the result is too large to be represented as a `float`, return `inf` if the original `float` was positive, and `-inf` if it was negative.

Once your program is working, you should see something like:

```
$ ./float_2048 1
2048
$ ./float_2048 3.14159265
6433.98193
$ ./float_2048 -2.718281828e-20
-5.56704133e-17
$ ./float_2048 1e38
inf
$ ./float_2048 -1e37
-inf
$ ./float_2048 inf
inf
$ ./float_2048 -inf
-inf
$ ./float_2048 nan
nan
```

Use *make* to build your code:

```
$ make    # or 'make float_2048'
```

HINT:

Some of the functions from the previous exercise may be useful.

Adding 11 to the exponent is the same as multiplying by 2048, but you will need to check this doesn't make the exponent too large for a float.

NOTE:

`float_2048` takes and returns a `uint32_t` that encode a single-precision floating-point number. You can assume the supplied `float` is not denormal.

You may define and call additional functions if you wish.

You are not permitted to call any functions from the C library, or to perform floating-point operations, or to use variables of type `float` or `double`, or to use multiplication or division.

You are not permitted to change the `main` function you have been given, or to change `float_2048`' prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest float_2048
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab03_float_2048 float_2048.c
```

You must run `give` before **Monday 05 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `float_2048.c`

Sample solution for float_2048.c

```c
// Sample solution for COMP1521 lab exercises
//
// Multiple a float by 2048 using bit operations only

#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

#include "floats.h"

float_components_t float_to_components(uint32_t f);
uint32_t float_from_components(float_components_t f);

#define EXPONENT_INF_NAN   0xFF

// float_2048 is given the bits of a float as a uint32_t
// it uses bit operations and + to calculate f * 2048
// and returns the bits of this value as a uint32_t
//
// if the result is too large to be represented as a float +inf or -inf is returned
//
// if the float is +inf or -int, or Nan it is returned unchanged
//
// float_2048 assumes the float is not a denormal number
//
uint32_t float_2048(uint32_t bits) {
    float_components_t f = float_to_components(bits);

    if (f.exponent != 0 && f.exponent != EXPONENT_INF_NAN) {
        // if we have a normal non-zero number increase the exponent

        f.exponent += 11;

        if (f.exponent >= EXPONENT_INF_NAN) {

            // number too large to be represented
            // set fraction & exponent to represent inf

            f.fraction = 0;
            f.exponent = EXPONENT_INF_NAN;
        }
    }

    return float_from_components(f);
}

#define SIGN_SHIFT       31
#define SIGN_MASK        0x1
#define EXPONENT_SHIFT   23
#define EXPONENT_MASK    0xFF
#define FRACTION_SHIFT   0
#define FRACTION_MASK    0x7FFFFF

// separate out the 3 components of a float
float_components_t float_to_components(uint32_t f) {
    float_components_t c;
    c.exponent = (f >> EXPONENT_SHIFT) & EXPONENT_MASK;
    c.fraction = (f >> FRACTION_SHIFT) & FRACTION_MASK;
    c.sign = (f >> SIGN_SHIFT) & SIGN_MASK;
    return c;
}


// build a float from its 3 components
uint32_t float_from_components(float_components_t f) {
    return (f.sign << SIGN_SHIFT)
         | (f.exponent << EXPONENT_SHIFT) |
           (f.fraction << FRACTION_SHIFT);
}
```

CHALLENGE EXERCISE — INDIVIDUAL:

Compare Floats Using Bit Operations

Your task is to add code to this function in **float_less.c**:

```
// float_less is given the bits of 2 floats bits1, bits2 as a uint32_t
// and returns 1 if bits1 < bits2, 0 otherwise
// 0 is return if bits1 or bits2 is Nan
// only bit operations and integer comparisons are used
uint32_t float_less(uint32_t bits1, uint32_t bits2) {
    // PUT YOUR CODE HERE

    return 42;
}
```

Add code to the function `float_less`, which takes the bits of two floats as the type `uint32_t`, and returns 1 if the value of the first `float` is less than the second, and 0 otherwise. It should do this using bit operations and integer comparisons.

For example:

```
$ ./float_less 1 2
float_less(1, 2) returned 1 which is correct
float_less(2, 1) returned 0 which is correct
$ ./float_less -3.14159265 -2.718281828e-20
loat_less(-3.14159274, -2.7182819e-20) returned 1 which is correct
float_less(-2.7182819e-20, -3.14159274) returned 0 which is correct
$ ./float_less -inf inf
float_less(-inf, inf) returned 1 which is correct
float_less(inf, -inf) returned 0 which is correct
$ ./float_less -0 0
float_less(-0, 0) returned 0 which is correct
float_less(0, -0) returned 0 which is correct
$ ./float_less NaN inf
float_less(nan, inf) returned 0 which is correct
float_less(inf, nan) returned 0 which is correct
```

Use *make* to build your code:

```
$ make    # or 'make float_less'
```

HINT:

The functions you wrote earlier in this lab may be useful.

Consider each of the cases carefully.

NOTE:

`float_less` should return an `int`, either 0 or 1.

You may define and call your own functions if you wish.

You can assume the supplied `floats` are not in denormal form.

You are not permitted to call any functions from the C library, or to perform floating-point operations, or to use variables of type `float` or `double`, or to use multiplication or division.

You are not permitted to change the `main` function you have been given, or to change `float_less`' prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest float_less
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab03_float_less float_less.c
```

You must run `give` before **Monday 05 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `float_less.c`

```c
// Sample solution for COMP1521 lab exercises
//
// Compare 2 floats using bit operations only

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

#include "floats.h"

float_components_t float_to_components(uint32_t f);
uint32_t float_from_components(float_components_t f);

// float_compare is given the bits of 2 floats f1, f2 as a uint32_t
// and returns 1 if f1 < f2, 0 otherwise
// 0 is return if f1 or f2 is Nan
// only bit operations and integer comparisons are used
uint32_t float_less(uint32_t bits1, uint32_t bits2) {
    float_components_t f1 = float_to_components(bits1);
    float_components_t f2 = float_to_components(bits2);

    if (is_nan(f1) || is_nan(f2)) {
        return 0;
    }

    if (is_negative_infinity(f1)) {
        // f1 is -inf, return 1 if f2 is not -inf
        return !is_negative_infinity(f2);
    }

    if (is_positive_infinity(f1) || is_negative_infinity(f2)) {
        return 0;
    }

    if (is_positive_infinity(f2)) {
        return 1;
    }

    if (is_zero(f1) && is_zero(f2)) {
        // special needed to handle +0, -0
        return 0;
    }

    if (f1.sign != f2.sign) {
        // f1, f2 have different signs, return 1 if f1 negative
        return f1.sign;
    }

    if (f1.exponent != f2.exponent) {
        // f1, f2 have different exponents
        if (f1.sign) {
            return f2.exponent < f1.exponent;
        } else {
            return f1.exponent < f2.exponent;
        }
    }

    // exponent and sign are identical
    if (f1.sign) {
        return f2.fraction < f1.fraction;
    } else {
        return f1.fraction < f2.fraction;
    }
}

#define SIGN_SHIFT       31
#define SIGN_MASK        0x1
#define EXPONENT_SHIFT   23
#define EXPONENT_MASK    0xFF
#define FRACTION_SHIFT   0
#define FRACTION_MASK    0x7FFFFF

#define EXPONENT_INF_NAN 0xFF
```

```c
// separate out the 3 components of a float
float_components_t float_to_components(uint32_t f) {
    float_components_t c;
    c.exponent = (f >> EXPONENT_SHIFT) & EXPONENT_MASK;
    c.fraction = (f >> FRACTION_SHIFT) & FRACTION_MASK;
    c.sign = (f >> SIGN_SHIFT) & SIGN_MASK;
    return c;
}


// build a float from its 3 components
uint32_t float_from_components(float_components_t f) {
    return (f.sign << SIGN_SHIFT)
         | (f.exponent << EXPONENT_SHIFT) |
           (f.fraction << FRACTION_SHIFT);
}

int is_nan(float_components_t f) {
    return f.exponent == EXPONENT_INF_NAN && f.fraction != 0;
}

int is_positive_infinity(float_components_t f) {
    return f.exponent == EXPONENT_INF_NAN && f.fraction == 0 && f.sign == 0;
}

int is_negative_infinity(float_components_t f) {
    return f.exponent == EXPONENT_INF_NAN && f.fraction == 0 && f.sign == 1;
}

int is_zero(float_components_t f) {
    return f.exponent == 0 && f.fraction == 0;
}
```

CHALLENGE EXERCISE — INDIVIDUAL:

Print A Float Using Only putchar and puts

Your task is to add code to this function in **float_print.c**:

```c
//
// float_print is given the bits of a float as a uint32_t
// it prints out the float in the same format as "%.9g\n"
// using only putchar & puts
//
void float_print(uint32_t bits) {
    // PUT YOUR CODE HERE
}
```

The function `float_print` takes a float as a `uint32_t`. Add code to it that prints its value as a decimal, in `"%.9g"` format, using only *putchar* or *puts*. You cannot use any other functions, such as *printf*.

Once your implementation is working, you should see things like:

```
$ ./float_print 1
1
$ ./float_print 3.14159265
3.14159274
$ ./float_print -2.718281828e-20
-2.7182819e-20
$ ./float_print 1.7e38
1.69999998e+38
$ ./float_print inf
inf
$ ./float_print -inf
-inf
$ ./float_print nan
nan
```

Use *make* to build your code:

```
$ make    # or 'make float_print'
```

HINT:

Some of the functions from the previous exercises may be useful.

You may find reading some of these papers helpful:

- David Goldberg's "What Every Computer Scientist Should Know About Floating-Point Numbers" from ACM Computing Surveys, March 1991;
- William D. Clinger's "How to Read Floating-Point Numbers Accurately" (doi:10.1145/93548.93557) from the proceedings of the 1990 ACM Conference on Programming Language Design and Implementation; and
- Guy L. Steele and Jon L. White's "How to Print Floating-Point Numbers Accurately" (doi:10.1145/93542.93559), also from the proceedings of the 1990 ACM Conference on Programming Language Design and Implementation.

NOTE:

You may define and call your own functions if you wish.

You can assume the supplied `float` is not in denormal form.

You are not permitted to call any functions from the C library, or to perform floating-point operations, or to use variables of type `float` or `double`.

You are not permitted to change the `main` function you have been given, or to change `float_print`'s prototype (its return type and argument types).

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest float_print
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab03_float_print float_print.c
```

You must run `give` before **Monday 05 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `float_print.c`

```c
//
// Sample solution for COMP1521 lab exercises
//
// Multiple a float by 2048 using bit operations only

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <assert.h>

void float_print(uint32_t f);

//
// We use this union to obtain the raw bits of a float
// by storing the float in the f field
// and then using the u field to obtain the bits
//

union overlay {
    float f;
    uint32_t u;
};

int main(int argc, char *argv[]) {
    for (int arg = 1; arg < argc; arg++) {
        union overlay input;

        input.f = atof(argv[arg]);
        float_print(input.u);
    }

    return 0;
}

// float_print is given the bits of a float as a uint32_t
// it prints out the float in the same format as "%.9g\n"
// using only putchar & puts
void float_print(uint32_t bits) {
    // cheating to produce autotest expected output
    union overlay input;
    input.u = bits;
    printf("%.9g\n", input.f);
}
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Mon Oct 5 21:00:00 2020** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```