NumCSE exercise sheet 0
Introduction to C++

alexander.dabrowski@sam.math.ethz.ch

September 29, 2018

Exercise 0.1. *Representing numbers in memory.*

Note: You can assume that `int` and `float` both occupy 32 bits in memory.

(a) How are integers represented in memory?

Solution: As a contiguous sequence of bits which represents the number in binary. If the integers can be negative, the easiest way to distinguish between positive and negative is to reserve a bit to represent the sign; however this is not the most convenient representation for computation purposes, and most computers store negative integers with the *two's complement* representation[1].

(b) Implement a function `void int_to_bits(int x)` which prints the bit representation of `x`.

Hints: Use the bitwise shift operator `<<` and the bitwise AND operator `&`.[2]

Solution:

```
void int_to_bits(int x) {
    for (int i = sizeof(int) * 8 - 1; i >= 0; i--)
        cout << (bool) (x & (1 << i));
}
```

to_binary.cpp

(c) How are floating point numbers represented in memory?

Solution: A floating point number is stored as a contiguous sequence of bits: one for the sign, a certain amount (`8` for `float`, `11` for `double`) for the binary representation of the exponent, and a certain amount (`23` for `float`, `52` for `double`) for the binary representation of the significant digits after the comma.[3]

(d) Implement a function `void float_to_bits(float x)` which prints the bit representation of `x`.

Hint: Casting a `float` to an `int` truncates the fractional part, but no information is lost casting a `float` pointer to an `int` pointer.

Solution:

[1] `https://en.wikipedia.org/wiki/Two's\_complement`.

[2] Given `char x` and `char n`, `x << n` shifts by `n` positions to the left the bits which represent `x`, while `x & n` applies bit per bit the AND operator. If for example the bits of `x` are `10100101`, then `x << 1` is `01001010`. If `y` is `00100000` then `x & y` is `00100000`. Also, recalling that any non-zero value cast to `bool` is converted to `1`, `(bool) (x & y)` is `1`. The same considerations hold for `int`, only in this case the bit representations have length 32 instead of 8.

[3] a very worthwhile read: David Goldberg, "What every computer scientist should know about floating-point arithmetic", ACM Computing Surveys, 23 (1): 5–48, (March 1991).

```cpp
void float_to_bits(float x) {
    assert(sizeof(int) == sizeof(float)); // to be sure that sizes actually match
    int* d = (int*) &x;
    int_to_bits(*d);
}
```

to_binary.cpp

Exercise 0.2. *Fast and accurate powers.*

In this exercise use only elementary functions (`+`, `-`, `*`, `/`, `%`, `&`, ...) or `std::exp`, `std::log` from `cmath`.

(a) Implement a function `double power(double a, int b)` which returns $\mathtt{a}^{\mathtt{b}}$.

Solution:

```
double power(double a, int b) {
    double tmp = 1;
    if (b < 0) {
        a = 1/a;
        b = -b;
    }
    while (b > 0) {
        tmp *= a;
        b--;
    }
    return tmp;
}
```

fast_powers.cpp

(b) Implement a function `double fast_power(double a, int b)` which returns $\mathtt{a}^{\mathtt{b}}$ in $O(\log \mathtt{b})$.

Hint: It might be useful to rewrite $\mathtt{a}^{\mathtt{b}}$ as $(\mathtt{a}^2)^{\mathtt{b}/2}$ when `b` is even and as $\mathtt{a}\,\mathtt{a}^{\mathtt{b}-1}$ when `b` is odd.

Solution:

```
double power_fast(double a, int b) {
    if (b < 0) {
        a = 1/a;
        b = -b;
    }
    if (b == 0)
        return 1;
    if (b == 1)
        return a;
    if (b & 1)
        return a * power_fast(a, b-1);
    return power_fast(a*a, b/2);
}
```

fast_powers.cpp

(c) Implement a function `double fast_power(double a, double b)` which returns $\mathtt{a}^{\mathtt{b}}$ in $O(1)$.

Hint: Use `std::exp` and `std::log`. You can assume that both run in $O(1)$.

Solution:

```
double power_fast(double a, double b) {
    return exp(b * log(a));
}
```

fast_powers.cpp

(d) Does any of your functions give exactly the same result of `std::pow(a,b)` for all inputs `a` and `b`? Why?

Solution: No. For big values of `b` the errors in the multiplications between floating point numbers propagate and reduce the accuracy of the results. The implementation of fast standard functions accurate up to the last significant digit requires a significant amount of work and attention.

(e) How big is the relative error between your implementations and `std::pow(a,b)` on average if `a` is an integer in $[1, 100)$ and `b` is an integer in $[0, 300)$? (discard the cases when either of your functions or `std::pow` overflow; you can use `std::isinf` to check if a `double` has overflown). Which one of your implementations minimizes the error? Why?

Solution:

```
int main() {
    double err1 = 0, err2 = 0, err3 = 0;
    int count = 0;
    for (double a = 1; a < 100; a++) {
        for (int b = 0; b < 300; b++) {
            if(isinf(power(a,b)) || isinf(power_fast(a,b)) ||
                isinf(power_fast(a, (double) b)) || isinf(pow(a,b)))
                continue;
            err1 += abs((power(a,b) - pow(a,b)) / pow(a,b));
            err2 += abs((power_fast(a,b) - pow(a,b)) / pow(a,b));
            err3 += abs((power_fast(a, (double) b) - pow(a,b)) / pow(a,b));
            count++;
        }
    }
    cout << err1/count << " " << err2/count  << " " << err3/count;
}
```

fast_powers.cpp

The output is: `3.00885e-16 1.7615e-16 1.9356e-14`

The relative errors are all not far from machine precision `EPS` $\simeq$ `2.22e-16`. In the first case the error is higher than in the second case, as we are doing more multiplications between floating point numbers ($\Theta($`b`$)$ instead of $\Theta(\log$ `b`$)$). The error is highest in the third case mainly due to the sensitivity to small changes of the exponential function.

Exercise 0.3. *Templates and factorials.*

(a) Implement a template function `factorial` which returns the factorial of an integer (of type `char`, or `int`, or `long`, ...) with the same return type as the input. Implement an iterative solution (no recursive calls).

Solution:

```
template<class T> T factorial(T x) {
    T out = 1;
    while(x > 1) {
        out *= x;
        x -= 1;
    }
    return out;
}
```

templates_factorials.cpp

(b) Implement a template function `dbl_factorial` which returns the factorial of any number as a `double` (with the convention that the factorial of a non-integer number is the factorial of its closest integer).

Hint: Use `std::round`.

Solution:

```
template<class T> double dbl_factorial(T x) {
    return factorial((double) round(x));
}
```

templates_factorials.cpp

(c) For which input values will your implementation of `factorial` return an accurate value if we pass an `int`? What if we pass a `long`? What if we pass a `double`?

Hint: If you include `climits` and `float.h` you can access the maximum `int` / `long` / `double` with the macro `INT_MAX` / `LONG_MAX` / `DBL_MAX`.

Solution:

```
template<class T> void test(T max) {
    // To find the maximum correct value for the factorial, consider the
    // maximum value of the type considered and divide it by
    // 2, 3, 4, ... until you get something smaller than 1.
    // Why this works? Hint: if a,b,c are integral types, then (a/b)/c == a/(b*c).
    T i = 1, x = max;
    while (x >= 1) {
        x /= ++i;
    }
    cout << "Factorial of " << i-1 << " is " << factorial(i-1)
         << ", but factorial of " << i << " is not " << factorial(i) << endl;
}

int main() {
    test(INT_MAX); test(LONG_MAX); test(DBL_MAX);
}
```

templates_factorials.cpp

Exercise 0.4. *(Optional) Dynamic array implementation.*

Implement a barebone `struct vec` to represent a dynamic array. The struct should have members:

- `capacity`, the maximum number of elements which can fit in the current instance of the array;
- `size`, the number of filled slots in the array;
- `double* data`, a pointer to an array of length `capacity`.

(a) Implement a default constructor which sets `capacity` to 10, `size` to 0, and allocates in `data` a new array of length 10.

(b) Implement a method `void push_back(double x)` which appends the element `x` to the dynamic array in amortized $O(1)$ time.

Hint: If `size < capacity` simply insert `x` at position `size` in `data`, and increase `size` by `1`. However if `size >= capacity`, first allocate a new array of doubled capacity, copy in it all the old values and reassign it to `data` (remember to `delete[]` the old array).

Solution:

```
struct vec {
    int capacity = 0;
    int size = 0;
    double* data = nullptr;

    vec() {
        capacity = 10;
        data = new double[capacity];
    }

    ~vec() {
        delete[] data;
    }

    void push_back(double x) {
        if (size >= capacity) {
            capacity *= 2;
            double* newdata = new double[capacity];
            for (int i=0; i < size; i++)
                newdata[i] = data[i];
            delete[] data;
            data = newdata;
        }
        data[size++] = x;
    }
};
```

dynamic_array.cpp