An Introduction to C Programming

Exercise Review and Discussion Session

David Dobrigkeit Chinellato
davidddc@ifi.unicamp.br

(Many thanks to Francesco Safai Tehrani!)
(these slides are based on his)

UNICAMP

Technicalities: How-to

- Compiler used: gcc ('pure C' + GCC extensions)
- Everything compiled via:
  - `gcc -Wall -pedantic [program] -o [executable`
  - `-Wall` = -W(arning) all
  - `-pedantic` = activate all checks for 'pure' C conformance

- no command line argument management
  - I basically ignore all the command line processing
  - arguments are 'passed' as #define-s

Some more details

- Techniques that were necessary here:
  – C program structure
    - How to organize your code into functioning pieces
  – Basic debugging
    - Printing out some messages (printf, etc)
  – Basic algorithmic thinking
    - Take an idea described in text and implement it!
  – Basic memory management
    - Declaring variables, arrays, allocating memory…

I will try to emphasize these when going through the exercises..

Questions? Please just ask!

‘Lost in translation’…

- Beware: **C ≠ C++**
  - …though, yes, they do look alike!

‘Lost in translation’…

- Beware: **C ≠ C++**
  - …though, yes, they do look alike!
- This is easy to mix… usually due to learning C++ but not ‘proper’ C. Some C++-isms are typically:
  - C++ style comments: //comment (instead of /* */)
    C++ memory management (new/delete, not malloc/free)
  - Arrays with variable size (actually, this is ISO C99, acepted by gcc as extension!). But in pure C, arrays must have constant size…

‘Lost in translation’…

- Beware: **C ≠ C++**
  - …though, yes, they do look alike!
- This is easy to mix… usually due to learning C++ but not ‘proper’ C. Some C++-isms are typically:
  - C++ style comments: //comment (instead of /* */)
    C++ memory management (new/delete, not malloc/free)
  - Arrays with variable size (actually, this is ISO C99, acepted by gcc as extension!). But in pure C, arrays must have constant size…
- Mixing is alright; mostly it will work. But…
  - You should know what you are doing! (e.g. use exactly the same (C++) compiler, or else care has to be taken!)

A bit of terminology, from C to C++

- Function **Interface** (or ‘signature’)
  - The set of **arguments** that a function accepts
  - E.g. `int sum(int a,int b)`
    - `“int a, int b”` is the interface of function ‘sum’

A bit of terminology, from C to C++

- Function **Interface** (or ‘signature’)
  - The set of **arguments** that a function accepts
  - E.g. `int sum(int a,int b)`
    - `“int a, int b”` is the interface of function ‘sum’
- A (collection of) header file(s): library interface
  - …the headers publish the interfaces

A bit of terminology, from C to C++

- Function **Interface** (or ‘signature’)
  - The set of **arguments** that a function accepts
  - E.g. `int sum(int a,int b)`
    - “`int a, int b`” is the interface of function ‘sum’
- A (collection of) header file(s): library interface
  - …the headers publish the interfaces
- Interfaces are very important !
  - …also in **O**bject **O**riented **P**rogramming in general!
  - More later…

Exercise Listing

Stuff we'll discuss the solutions to

1. Forward-counting factorial
2. Fibonacci Numbers
3. Unit Conversion Library
4. Crash the stack
5. Returning Multiple Values
6. Numeric Integration
7. Endianness
8. Integration with Monte Carlo Method
9. 1D Cellular Automata
10. The Sieve of Erastothenes

If you want to take a look:

https://www.dropbox.com/sh/8jilt0ngxdfgrte/AACv9t5rMW7-SPax3K7xNRG9a?dl=0

It is important to understand the ideas! (Code is available online, knowledge isn't!)

Ex. 1: Factorial

Rewrite both the iterative and recursive factorial functions using forward counting (from 1 to num).

Rather simple to implement, both recursively and iteratively…

Ex. 1: Factorial

Rewrite both the iterative and recursive factorial functions using forward counting (from 1 to num).

Rather simple to implement, both recursively and iteratively…

But: we asked you to implement the algorithm with forward-counting! This requires that the recursive version gets a second parameter (a 'multiplication counter', if you will), which the user will **not care about**. We should have:

```
factorial(n)   →   factorial(n , counter)
```

…but **if the user doesn't care**, do we want to **publish** the interface 'n, counter'? How do we solve this issue?

Ex. 1: Factorial

Rewrite both the iterative and recursive factorial functions using forward counting (from 1 to num).

Rather simple to implement, both recursively and iteratively…

But: we asked you to implement the algorithm with forward-counting! This requires that the recursive version gets a second parameter (a 'multiplication counter', if you will), which the user will **not care about**. We should have:

`factorial(n)` → `factorial(n , counter)`

…but **if the user doesn't care**, do we want to **publish** the interface 'n, counter'? How do we solve this issue?

Let's use the **"Principle of least surprise"!**

Ex. 1: Factorial

```c
#include <stdio.h>

int recursive_forward_factorial(int num, int index) {
  if(index==num) return num;
  return index * recursive_forward_factorial(num, index+1);
}
```

Auxiliary function (never 'seen' by end user!)

```c
inline int recursive_factorial(int num) {
  return recursive_forward_factorial(num, 1);
}
```

Function with expected interface

```c
int iterative_factorial(int num) {
     int i, result=1;
     for (i=1; i<=num; i++) result *= i;
     return result;
}

int main (int argc, const char * argv[]) {
     printf("Iterative 6! :%d\n", iterative_factorial(6));
     printf("Recursive 6! :%d\n", recursive_factorial(6));
     return 0;
}
```

Ex. 2: Fibonacci Numbers

Write a program containing two functions (iterative and recursive) which calculate the *n*-th Fibonacci number, defined as:

$$F_{n+2} = F_{n+1} + F_n \text{, with } F_0 = 0 \text{ and } F_1 = 1$$

Another relatively simple task, but one has to keep track of **the last two** results.

Ex. 2: Fibonacci Numbers

Write a program containing two functions (iterative and recursive) which calculate the *n*-th Fibonacci number, defined as:

$F_{n+2} = F_{n+1} + F_n$, with $F_0 = 0$ and $F_1 = 1$

Another relatively simple task, but one has to keep track of **the last two** results.

However, the recursive implementation is not efficient! Why?

Ex. 2: Fibonacci Numbers

Write a program containing two functions (iterative and recursive) which calculate the *n*-th Fibonacci number, defined as:

$F_{n+2} = F_{n+1} + F_n$, with $F_0 = 0$ and $F_1 = 1$

Another relatively simple task, but one has to keep track of **the last two** results.

However, the recursive implementation is not efficient! Why?

- Because each function call is transformed into two! This seems to indicate that the algorithmic complexity will be $\mathbf{2^n}$ and will thus be very slow…
- In reality, it is a bit better than that, but to put this in perspective, F_{49} would require 49 sums if done iteratively, but $\sim 10^{10}$ if done recursively…

Ex. 2: Fibonacci Numbers

Write a program containing two functions (iterative and recursive) which calculate the *n*-th Fibonacci number, defined as:

$$F_{n+2} = F_{n+1} + F_n \text{ , with } F_0 = 0 \text{ and } F_1 = 1$$

Another relatively simple task, but one has to keep track of **the last two** results.

However, the recursive implementation is not efficient! Why?

- Because each function call is transformed into two! This seems to indicate that the algorithmic complexity will be $\mathbf{2^n}$ and will thus be very slow…
- In reality, it is a bit better than that, but to put this in perspective, F_{49} would require 49 sums if done iteratively, but ~10^{10} if done recursively…
  - **(can we be smarter than this?)**

Ex. 2: Fibonacci Numbers

```c
#include <stdio.h>

#define FIBNUM 49 /* the highest Fn found on the Wikipedia page */

long recursive_fibonacci(int num) {
  if (num<=1) return num;
  return recursive_fibonacci(num-1)+recursive_fibonacci(num-2);
}

long iterative_fibonacci(int num) {
    int i;
    long f0, f1, tmp;
    if (num<=1) return num;

    f0 = 0;
    f1 = 1;
    for (i=2; i<=num; i++) {
      tmp = f0;
      f0 = f1;
      f1 = f0+tmp;
    }
    return f1;
}

int main (int argc, const char * argv[]) {
    printf("The %d Fibonacci number is (iteratively) :%ld\n", FIBNUM, iterative_fibonacci(FIBNUM));
    printf("The %d Fibonacci number is (recursively) :%ld\n", FIBNUM, recursive_fibonacci(FIBNUM));
    return 0;
}
```

Guilty as charged: Double recursion!

This will be fast (<0.001s)

This will be slow! (60s or so…)

Ex. 2: Fibonacci Numbers

```
#include <stdio.h>

#define FIBNUM 49 /* the highest Fn found on the Wikipedia page */

long recursive_fibonacci(int num) {
  if (num<=1) return num;
  return recursive_fibonacci(num-1)+recursive_fibonacci(num-2);
}

long iterative_fibonacci(int num) {
    int i;
    long f0, f1, tmp;
    if (num<=1) return num;

    f0 = 0;
    f1 = 1;
    for (i=2; i<=num; i++) {
      tmp = f0;
      f0 = f1;
      f1 = f0+tmp;
    }
    return f1;
}

int main (int argc, const char * argv[]) {
    printf("The %d Fibonacci number is (iteratively) :%ld\n", FIBNUM, iterative_fibonacci(FIBNUM));
    printf("The %d Fibonacci number is (recursively) :%ld\n", FIBNUM, recursive_fibonacci(FIBNUM));
    return 0;
}
```

Guilty as charged:
Double recursion!

This will be fast (<0.001s)

This will be slow! (60s or so…)

Alright…. But can we be smarter?

(yes, we can!)

Ex. 2: Fibonacci Numbers (2.0)

Let's recycle: No need to recompute if already done.
Needs storage! Here: global variables (nasty, but… hey, it works)

```
#include <stdio.h>

#define MAXFNS 100
long fns[MAXFNS];

#define FIBNUM 49 /* the last Fn found on the Wikipedia page */

long recursive_fibonacci(long num) {
  if (num<=1) return num;
  if (fns[num] == -1) {
    fns[num] = recursive_fibonacci(num-1)+recursive_fibonacci(num-2);
  }
  return fns[num];
}

int main (int argc, const char * argv[]) {
    int idx;
    for(idx=0; idx<MAXFNS; idx++) fns[idx] = -1;
    printf("The %d Fibonacci number is (recursively) :%ld\n", FIBNUM, recursive_fibonacci(FIBNUM));
    return 0;
}
```

This is what is called **'memoization'** *(no, this is not a typo, I also cross-checked, no missing 'r' :-D)*

What we've done is to replace expensive function calls by caching the result and returning directly from that cache if asked again. Here, the 'cache' is a global variable (not ideal…). This cache goes by many names (e.g. 'lookup table')

Ex. 2: Fibonacci Numbers (2.0)

```
#include <stdio.h>

#define MAXFNS 100
long fns[MAXFNS];

#define FIBNUM 49 /* the last Fn found on the Wikipedia page */

long recursive_fibonacci(long num) {
  if (num<=1) return num;
  if (fns[num] == -1) {
    fns[num] = recursive_fibonacci(num-1)+recursive_fibonacci(num-2);
  }
  return fns[num];
}

int main (int argc, const char * argv[]) {
    int idx;
    for(idx=0; idx<MAXFNS; idx++) fns[idx] = -1;
    printf("The %d Fibonacci number is (recursively) :%ld\n", FIBNUM, recursive_fibonacci(FIBNUM));
    return 0;
}
```

Let's recycle: No need to recompute if already done.
Needs storage! Here: global variables (nasty, but… hey, it works)

Only compute the first time!

This is what is called **'memoization'** *(no, this is not a typo, I also cross-checked, no missing 'r' :-D)*

What we've done is to replace expensive function calls by caching the result and returning directly from that cache if asked again. Here, the 'cache' is a global variable (not ideal…). This cache goes by many names (e.g. 'lookup table')

Ex. 2: Fibonacci Numbers (3.0)

```
#include<stdio.h>

typedef struct fibpair {
    long val, prevval;
} fibpair;

fibpair recfib(long num) {
    float sum;
    fibpair tmp = { 1, 0 };
    if (num == 1) return tmp;
    if (num == 0) {
        tmp.val = 0;
        return tmp;
    }
    tmp = recfib( num-1 );
    sum = tmp.val + tmp.prevval;
    tmp.prevval = tmp.val;
    tmp.val = sum;
    return tmp;
}

long recursive_fib(long num) {
    fibpair tmp = recfib( num );
    return tmp.val;
}

int main() {
    int i;
    printf("The %2ith Fib. number is = %ld\n", 49, recursive_fib(49));
    return 0;
}
```

It doesn't stop there: we can also store the added information via a C struct to hold the current and previous Fibonacci number. The data structure is more complex, but in this way no double recursion is needed!

N.B. This code is slightly less transparent… Another option would be to store the 'cache' from the memoization in a struct, too!

Ex. 3: Unit Conversion Library

- For this exercise you will implement a few unit conversion libraries. You can find the conversion factors and algorithms online.

- **[length]** Start with **a <u>library to convert centimeters to inches</u>**, meters to feet and vice versa, then add miles to kilometers. Add as many as you want.
- **[weight]** Now create another **<u>library to convert weights</u>**, and implement the conversion between kilograms and pounds. Add as many as you want.
- **[temperature]** Now create a library to **<u>convert between different temperature scales</u>**, Celsius to Fahrenheit and vice versa, Celsius to Kelvin, Kelvin to Fahrenheit and so on.

- Create a test program to use these libraries and print various conversions. Check that the result are correct. Now, unless you've done some design in advance, you will find yourself with a lot of functions which do exactly the same thing (more or less).

Ex. 3: Unit Conversion Library

- For this exercise you will implement a few unit conversion libraries. You can find the conversion factors and algorithms online.

- **[length]** Start with **a <u>library to convert centimeters to inches</u>**, meters to feet and vice versa, then add miles to kilometers. Add as many as you want.
- **[weight]** Now create another **<u>library to convert weights</u>**, and implement the conversion between kilograms and pounds. Add as many as you want.
- **[temperature]** Now create a library to **<u>convert between different temperature scales</u>**, Celsius to Fahrenheit and vice versa, Celsius to Kelvin, Kelvin to Fahrenheit and so on.

- Create a test program to use these libraries and print various conversions. Check that the result are correct. Now, unless you've done some design in advance, you will find yourself with a lot of functions which do exactly the same thing (more or less).

- **[utility]** Would it be possible to **<u>rewrite your conversion libraries to minimize code repetition</u>**, maybe by implementing some utility functions in a special dedicated library? (Utility functions are functions which solve a specific problem in a more general way).
- Rewrite your libraries to maximize code reuse. Is it simpler now to add new conversions? Discuss your solution.

Ex. 3: Unit Conversion Library

Length

Header

```c
#ifndef LENGTH_H
#define LENGTH_H

double cm_to_in(double);
double in_to_cm(double);

#endif
```

```c
#include "length.h"

double cm_to_in(double cms) {
  return cms/2.54;
}

double in_to_cm(double ins) {
  return ins*2.54;
}
```

Weight

Header

```c
#ifndef WEIGHT_H
#define WEIGHT_H

double kg_to_lb(double);
double lb_to_kg(double);

#endif
```

```c
#include "weight.h"

double kg_to_lb(double kgs) {
  return kgs/0.45359237;
}

double lb_to_kg(double lbs) {
  return lbs*0.45359237;
}
```

Temperature

Header

```c
#ifndef TEMPERATURE_H
#define TEMPERATURE_H

double C_to_F(double);
double F_to_C(double);

#endif
```

```c
#include "temperature.h"

double C_to_F(double cdeg) {
  return cdeg*9./5.+32.;
}

double F_to_C(double fdeg) {
  return (fdeg-32)*5./9.;
}
```

```c
#include <stdio.h>
#include "weight.h"
#include "temperature.h"
#include "length.h"

int main (int argc, const char * argv[]) {
  printf("1 cm in inches: %7.3f\n", cm_to_in(1.));
  printf("1 inch in   cms: %7.3f\n", in_to_cm(1.));
  printf("1 pound in kgs: %7.3f\n", lb_to_kg(1.));
  printf("1 kg in pounds: %7.3f\n", kg_to_lb(1.));
  printf("60F in Celsius: %7.3f\n",   C_to_F(60));
  printf("60Celsius in F: %7.3f\n",   F_to_C(60));
  return 0;
}
```

How can we make life easier?
Note that the conversions are always of the type:

y = A × x + B

…but then why not put this into a helper/utility library?

Ex. 3: Unit Conversion Library

Length

```c
#ifndef WEIGHT_H
#define WEIGHT_H

#include "utility.h"

double kg_to_lb(double);
double lb_to_kg(double);

#endif
```

```c
#include "weight.h"

double kg_to_lb(double kgs) {
  return reverse_conversion(kgs, 0.45359237, 0.);
}

double lb_to_kg(double lbs) {
  return direct_conversion(lbs, 0.45359237, 0.);
}
```

Weight

```c
#ifndef WEIGHT_H
#define WEIGHT_H

#include "utility.h"

double kg_to_lb(double);
double lb_to_kg(double);

#endif
```

```c
#include "weight.h"

double kg_to_lb(double kgs) {
  return reverse_conversion(kgs, 0.45359237, 0.);
}

double lb_to_kg(double lbs) {
  return direct_conversion(lbs, 0.45359237, 0.);
}
```

Temperature

```c
#ifndef TEMPERATURE_H
#define TEMPERATURE_H

#include "utility.h"

double C_to_F(double);
double F_to_C(double);

#endif
```

```c
#include "temperature.h"

double C_to_F(double cdeg) {
  return direct_conversion(cdeg, 9./5., 32.);
}

double F_to_C(double fdeg) {
  return reverse_conversion(fdeg, 9./5., 32.);
}
```

Utility

```c
#ifndef UTILITY_H
#define UTILITY_H

double direct_conversion(double, double, double);
double reverse_conversion(double, double, double);

#endif
```

```c
#include "utility.h"

double direct_conversion(double value, double m_factor, double a_factor) {
  return value*m_factor + a_factor;
}

double reverse_conversion(double value, double m_factor, double a_factor) {
  return (value-a_factor)/m_factor;
}
```

This maximizes code reuse!

It's good for any **linear** conversion, and it is enough to know the two coefficients involved to implement anything new.

The good old 'main' we showed in the previous slide wouldn't know the difference, of course!

Ex. 4: Crashing the stack (because breaking things is easy!)

- Write a program to crash the stack.
- As a bonus point, add a counter to check the stack depth.

That's an easy one! **Breaking things is easy, indeed …**

What needs to be done here is to issue function calls within function calls until the stack is exhausted and we get a crash.

- Let's count how long (how many function calls!) it takes, while we're at it!

Ex. 4: Crashing the stack (because breaking things is easy!)

```c
#include<stdio.h>

void crash_stack(int index) {
    printf("%d ... ", index);
    crash_stack(index+1);
}

int main() {
    crash_stack(0);
    return 0;;
}
```

For me, this crashes after 261818 function calls, with a message of:

```
Segmentation fault
(core dumped)
```

Ex. 5: Returning Multiple Values

- **[multiple value return]** Write a function that accepts two positive numbers, and returns their sum, their difference and their mean value. **[error handling]** Also make it so that the function returns something indicating an error if one of the arguments is negative.
- Write a program to use this function and print its results.

Ex. 5: Returning Multiple Values

- **[multiple value return]** Write a function that accepts two positive numbers, and returns their sum, their difference and their mean value. **[error handling]** Also make it so that the function returns something indicating an error if one of the arguments is negative.
- Write a program to use this function and print its results.

We can solve this by using two different approaches:

- Write a function with a long signature containing **three dummy arguments** passed by pointer, so that the function can store the values there
- Write a function that **returns an array** containing the results

Ex. 5: Returning Multiple Values

- **[multiple value return]** Write a function that accepts two positive numbers, and returns their sum, their difference and their mean value. **[error handling]** Also make it so that the function returns something indicating an error if one of the arguments is negative.
- Write a program to use this function and print its results.

We can solve this by using two different approaches:

- Write a function with a long signature containing **three dummy arguments** passed by pointer, so that the function can store the values there
- Write a function that **returns an array** containing the results

Most commonly, the first approach is used (also makes it easier to not confuse the array positions!). But the error handling is important! Better to return error values rather than print out an error message. Actually, error handling is much easier in the first approach… so let's focus on that!

Ex. 5: Returning Multiple Values

Dummy arguments (to be used as output)

Handling the error condition (negative input!)

```c
#include <stdio.h>

int calc_data(int op1, int op2, int* sum, int* diff, float* mean) {
  if((op1<0) || (op2<0)) { return -1; }
  *sum = op1+op2;
  *diff = op1-op2;
  *mean = *sum / 2.0;
  return 0;
}

int main (int argc, const char * argv[]) {
    int op1 = 40;
    int op2 = 2;

    int sum, diff;
    float mean;

    if(calc_data(op1, op2, &sum, &diff, &mean) == -1) {
        printf("Whoops! One of the operands of calc_data is negative!\n");
    } else {
        printf("Calc data results: \n");
        printf("              %5d + %5d = %5d\n", op1, op2, sum);
        printf("              %5d - %5d = %5d\n", op1, op2, diff);
        printf("Mean value of %5d,  %5d = %8.2f\n", op1, op2, mean);
    }
    return 0;
}
```

```
The printf formatting works like this:
%[-c]n[.m]X
The - specifies that the field is left justified
The [c] is a padding character: %05d, 42 = 00042
The n is the field length in characters
The [.m] only for float/double fields, indicates the number of digits after the '.'
X is the format specifier [d = integers, f = floats/doubles, ...]
```

Ex. 6: Numerical Integration

Write a program to calculate a numeric integral using the the composite trapezoidal rule (http://en.wikipedia.org/wiki/Trapezoidal_rule).

[Integrator] The program should define a function that accepts an array containing the values of the function to integrate and any other relevant parameter: float integrate(float values[], ...)

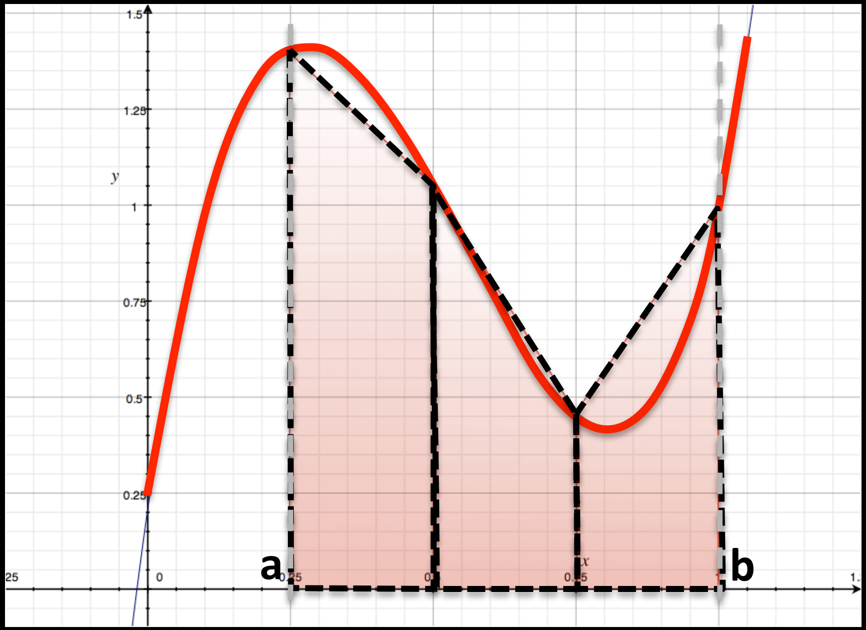
[main program] The main part of the program should fill the values array, with values calculated from the function to be integrated. Ideally this function should also be stored in a function (okay, the mathematical function to be integrated should be stored in a C function).

- This logical separation allows you to write the integration code and reuse it as needed, while making it also possible to easily implement other integration algorithms and reuse the same mathematical functions. You should be careful when defining the integration interval, the integration steps and all the relevant parameters. You might also want to define some utility function to map the integer indexes of the values array onto the integration step. The math.h header contains a number of mathematical functions which might be useful.

Ex. 6: Numerical Integration

The Trapezoidal Rule:
Take f(x), an interval [a, b] and divide it in N subintervals of length = (b-a)/N. Then:

$$\begin{cases} x_1 = a \\ \dots \\ x_i = a + i \times (\text{step length}) \\ \dots \\ x_N = b \end{cases}$$

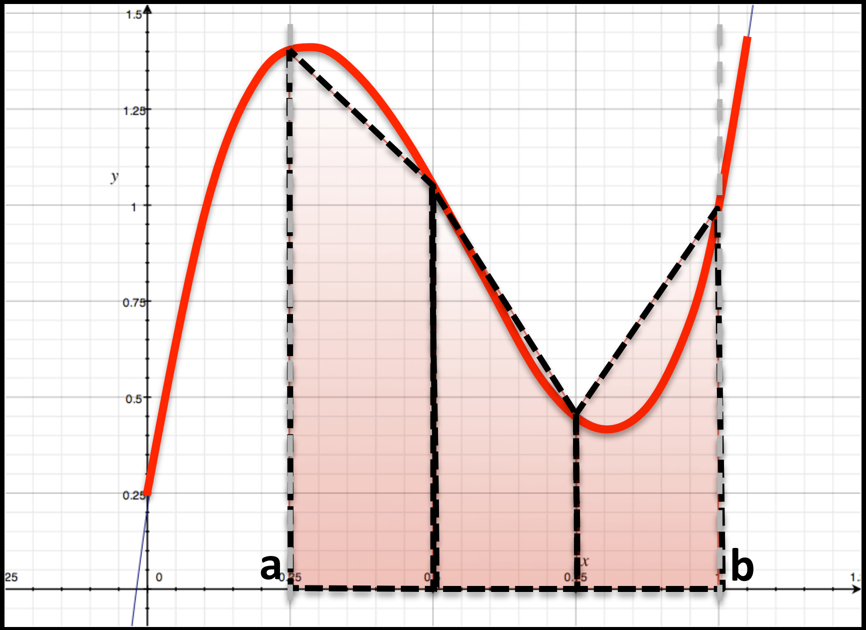


$$A = \sum \frac{1}{2} \left(f\left(x_i\right) + f\left(x_{i+1}\right) \right) \times \left(step\, length \right)$$

Ex. 6: Numerical Integration

The Trapezoidal Rule:
Take f(x), an interval [a, b] and divide it in N subintervals of length = (b-a)/N. Then:

$$\begin{cases} x_1 = a \\ \ldots \\ x_i = a + i \times (\text{step length}) \\ \ldots \\ x_N = b \end{cases}$$



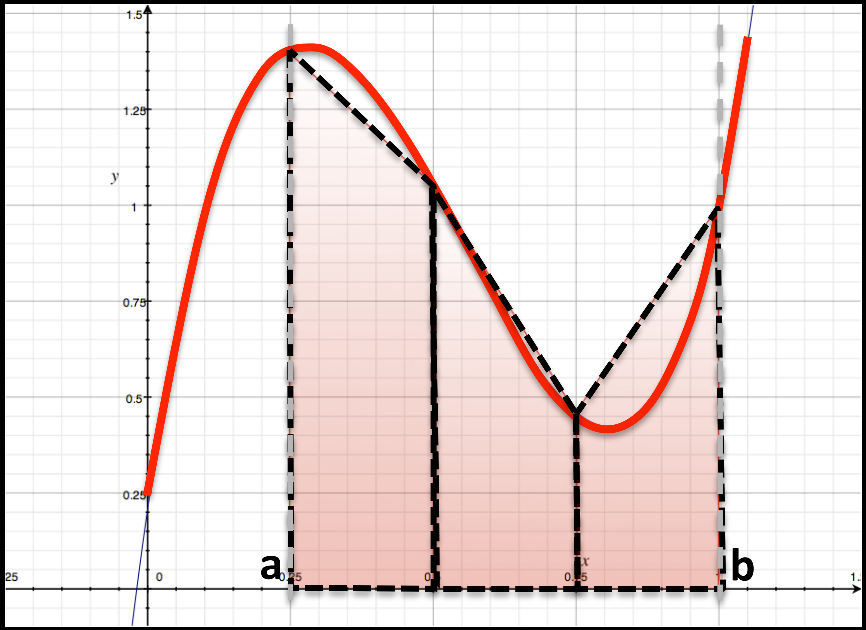
$$A = \sum \frac{1}{2} \left(f(x_i) + f(x_{i+1}) \right) \times (step\,length)$$

Think before coding! It pays off to implement a faithful version of the algorithm and not try to optimize too much beforehand.

Ex. 6: Numerical Integration

The Trapezoidal Rule:
Take f(x), an interval [a, b] and divide it in N subintervals of length = (b-a)/N. Then:

$$\begin{cases} x_1 = a \\ \dots \\ x_i = a + i \times (\text{step length}) \\ \dots \\ x_N = b \end{cases}$$



$$A = \sum \frac{1}{2} \left(f(x_i) + f(x_{i+1}) \right) \times (step\, length)$$

Think before coding! It pays off to implement a faithful version of the algorithm and not try to optimize too much beforehand.

In this case, without loss of generality, note that you can rewrite the sum a bit!

$$A = \left[\frac{1}{2} \left(f(a) + f(b) \right) + \sum_{i=1}^{N-1} f(x_i) \right] \times (step\ length)$$

Ex. 6: Numerical Integration

```c
#include <stdio.h>
#include <math.h>

#define STEPS 1000

double function(double x) {
    return exp(x)*pow(x,2);
}

void calculate_function(double a, double b, int steps, double* values) {
  int idx;
  double step_len = (b-a)/steps;
  for(idx=0; idx<=steps; idx++) {
    values[idx] = function(a+idx*step_len);
  }
}

double integrate_function(double a, double b, int steps, double* values) {
  int idx;
  double step_len = (b-a)/steps;
  double result = 0;

  for(idx=1; idx<steps; idx++)
    result += values[idx];

  result = (result + (values[0]+values[steps])/2.0) * step_len;
  return result;
}

int main (int argc, const char * argv[]) {
    double a, b;
    double result;
    double values[STEPS+1];

    a = 1;
    b = 2;

    calculate_function(a, b, STEPS, values);
    result = integrate_function(a, b, STEPS, values);

    printf("Integration of e^x*x^2 between %7.2f and %7.2f yields %7.2f\n", a, b, result);
    return 0;
}
```

Compute the values of the (mathematical!) function

Compute the integral using the trapezoidal Rule (modular!)

Main

Ex. 7: Endianness

- Figure out the endianness of the computer you are using using a C program

You can choose a short for simplicity! It only contains two bytes so that:

```
0x1 in little endian would look like 01 00
0x1 in big endian would look like    00 01
```

If you read the material suggested for the exercise, you should also know a little bit more about how C represents integer numbers internally.

Ex. 7: Endianness

- Figure out the endianness of the computer you are using using a C program

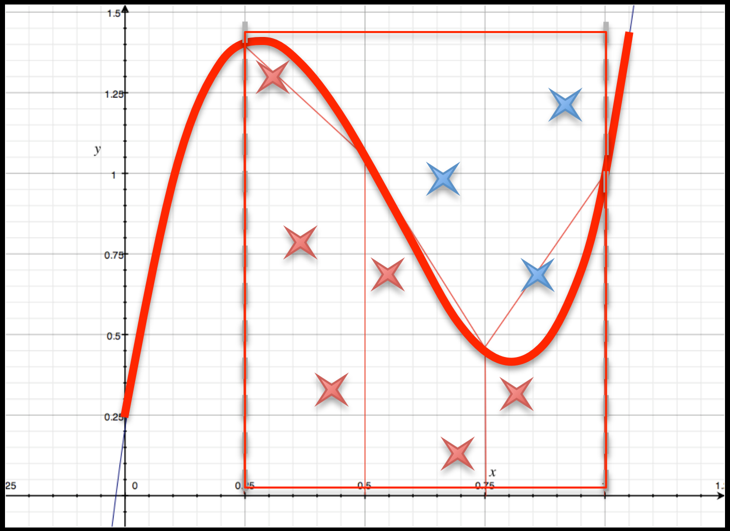
You can choose a short for simplicity! It only contains two bytes so that:

```
0x1 in little endian would look like 01 00
0x1 in big endian would look like    00 01
```

If you read the material suggested for the exercise, you should also know a little bit more about how C represents integer numbers internally.

```c
#include<stdio.h>

int main() {
    short a = 1;
    char* x = (char*) &a;
    if (x[0] == 1 ) { printf("Little endian\n"); }
    else            { printf("Big endian\n");    }
    return 0;
}
```

Ex. 8: Monte Carlo Integration



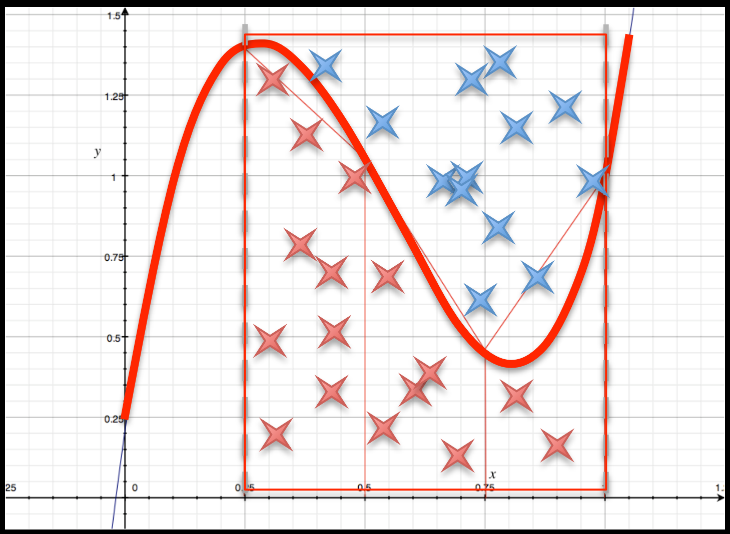
Write a program to implement the 1D Monte Carlo integration. The Monte Carlo integration is a numerical integration algorithm that uses random numbers. The algorithm works as follows:

1. Inscribe your function in a rectangle whose left and right sides are the same as the integration limits, and whose lower side lays on the x axis
2. Generate a random point within this rectangular area
3. If the point is under the curve, increment a counter
4. Repeat 2. and 3. N times. With N large enough, the integral of the curve is

(counter/N)*rectangle_area

See: http://en.wikipedia.org/wiki/Monte_Carlo_integration and man rand for information about random number generation in C.

Ex. 8: Monte Carlo Integration



Write a program to implement the 1D Monte Carlo integration. The Monte Carlo integration is a numerical integration algorithm that uses random numbers. The algorithm works as follows:

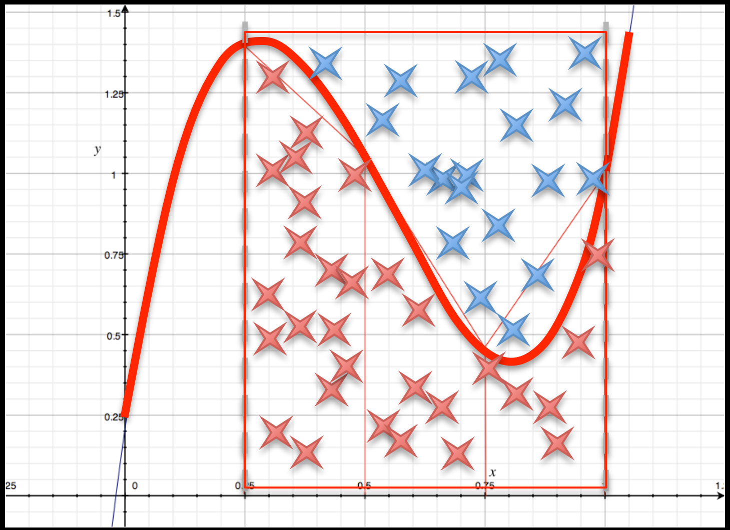
1. Inscribe your function in a rectangle whose left and right sides are the same as the integration limits, and whose lower side lays on the x axis
2. Generate a random point within this rectangular area
3. If the point is under the curve, increment a counter
4. Repeat 2. and 3. N times. With N large enough, the integral of the curve is

(counter/N)*rectangle_area

See: http://en.wikipedia.org/wiki/Monte_Carlo_integration and man rand for information about random number generation in C.

Ex. 8: Monte Carlo Integration

N_{above}



N_{below}

Area = (N_{below}/N_{all})× $A_{rectangle}$

Write a program to implement the 1D Monte Carlo integration. The Monte Carlo integration is a numerical integration algorithm that uses random numbers. The algorithm works as follows:

1. Inscribe your function in a rectangle whose left and right sides are the same as the integration limits, and whose lower side lays on the x axis
2. Generate a random point within this rectangular area
3. If the point is under the curve, increment a counter
4. Repeat 2. and 3. N times. With N large enough, the integral of the curve is

(counter/N)*rectangle_area

See: http://en.wikipedia.org/wiki/Monte_Carlo_integration and man rand for information about random number generation in C.

Ex. 8: Monte Carlo Integration

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define STEPS 1000
#define SAMPLES 10000000

double function(double x) { return exp(x)*pow(x,2); }

void min_max(double a, double b, int steps, double* min, double* max) {
  int idx;
  double value;
  double step_len = (b-a)/steps;

  *min = function(a);
  *max = function(a);

  for(idx=0; idx<=steps; idx++) {
    value = function(a+idx*step_len);
    if(value > *max) *max = value;
    if(value < *min) *min = value;
  }
}
```

First step: determine maximum and minimum of function within interval (so as to determine the rectangle to inscribe your function in!)

Ex. 8: Monte Carlo Integration

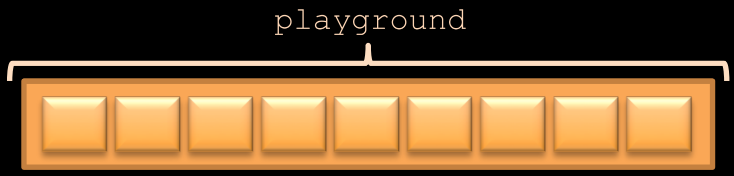
```
double mc_integrate_function(double a, double b, double min, double max, int samples) {
  int idx, counter = 0;
  double xr, yr;
  double lenH = b-a;
  double lenV = (max - min);

  srand(time(0));

  for(idx=0; idx<samples; idx++) {
    xr = a   + lenH * (rand()/(double)RAND_MAX);
    yr = min + lenV * (rand()/(double)RAND_MAX);

    if(function(xr) > yr) counter++;
  }
  return lenH*lenV*counter/samples;
}

int main (int argc, const char * argv[]) {
    double a, b, min, max, result;

    a = 1;
    b = 2;

    min_max(a, b, STEPS, &min, &max);
    if(min > 0) min = 0;
    if(max < 0) return 1;

    result = mc_integrate_function(a, b, min, max, SAMPLES);

    printf("Integration of e^x*x^2 between %7.2f and %7.2f yields %7.2f\n", a, b, result);
    return 0;
}
```

Function to "shoot randomly" within that triangle and count (fraction of) points below curve

Main Program

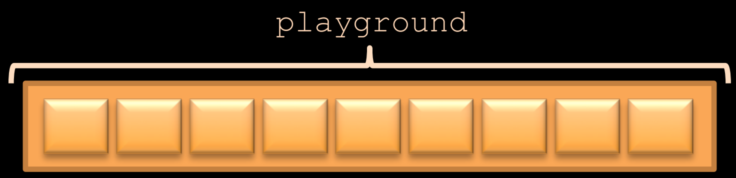
Ex. 9: 1D Cellular Automata

- Let's think of a **playground of N spaces.**



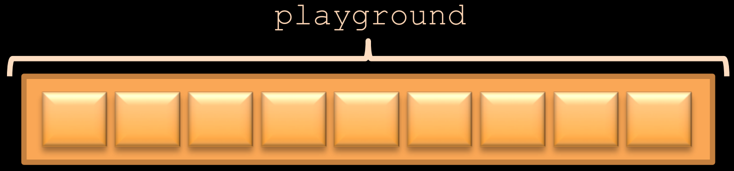
Ex. 9: 1D Cellular Automata

- Let’s think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent



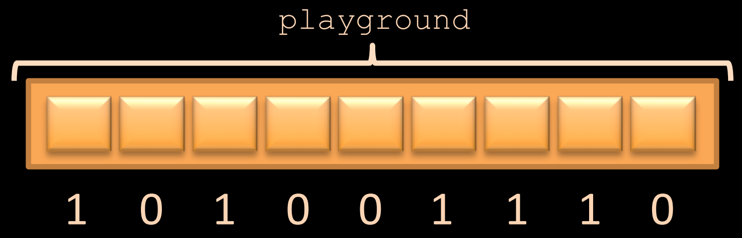
Ex. 9: 1D Cellular Automata

- Let’s think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**



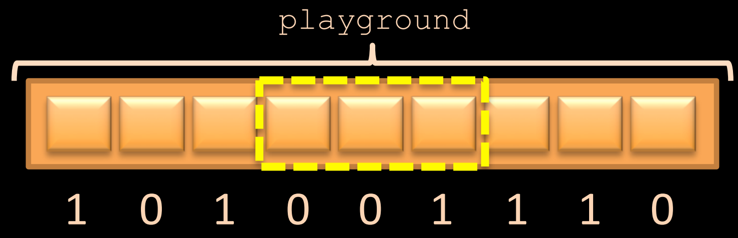
Ex. 9: 1D Cellular Automata

- Let’s think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state ‘0’**) or alive (**state ‘1’**)



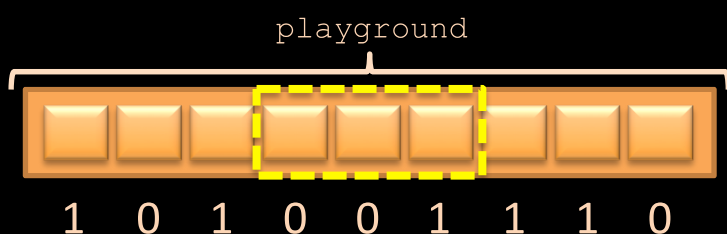
Ex. 9: 1D Cellular Automata

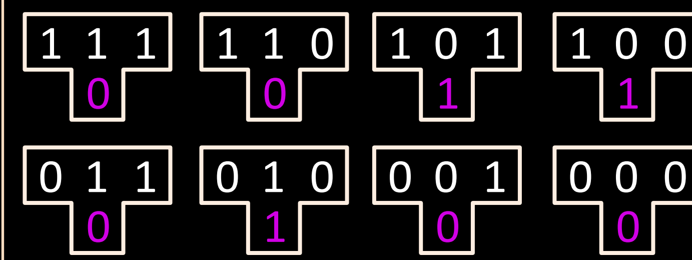
- Let’s think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state ‘0’**) or alive (**state ‘1’**)
- **Time flows in discrete steps;** at each given step the cell state can be updated, either swapping state or staying the same, depending on its neighbor cells (‘neighborhood’)



Ex. 9: 1D Cellular Automata

- Let's think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state '0'**) or alive (**state '1'**)
- **Time flows in discrete steps;** at each given step the cell state can be updated, either swapping state or staying the same, depending on its neighbor cells ('neighborhood')
- There are **only eight possible configurations** for a neighborhood, and each with a possible outcome of 0 or 1 in the next state. Thus, there are only 256 possible evolution rules! →

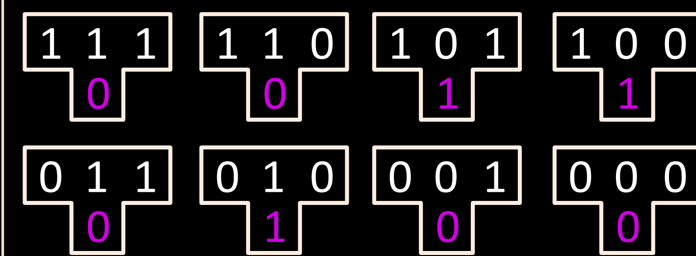




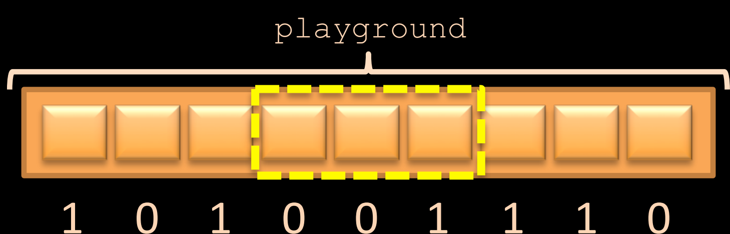
As an example:
This is rule 52 ($= 2^2 + 2^4 + 2^5$)

Ex. 9: 1D Cellular Automata

- Let's think of a **playground of N spaces.**
- **The playground is circular**, i.e. the leftmost and rightmost elements are adjacent
- **Each space** in the playground **is** called **a cell**
- Each cell is either dead (**state '0'**) or alive (**state '1'**)
- **Time flows in discrete steps;** at each given step the cell state can be updated, either swapping state or staying the same, depending on its neighbor cells ('neighborhood')
- There are **only eight possible configurations** for a neighborhood, and each with a possible outcome of 0 or 1 in the next state. Thus, there are only 256 possible evolution rules! →



As an example:
This is rule 52 ($= 2^2 + 2^4 + 2^5$)



This exercise will be discussed in more detail at 17:00 in the C++ Introduction!

Ex. 9: 1D Cellular Automata

```c
#include <stdio.h>
#include <stdlib.h>

#define PLAYGROUND_SIZE 80
#define GENERATIONS     20
#define RULE            30

void init_rules(int rule, int* evolutionTable) {
  int idx;
  for(idx=0; idx<8; idx++) {
    evolutionTable[idx] = (rule >> idx) & 1;
  }
}

void init_playground(int size, int* playground) {
  int idx;
  for(idx=0;idx<size; idx++) { playground[idx] = 0; }
  /* impulse */
  playground[size/2] = 1;
}

void print_playground(int size, int* playground) {
  int idx;
  for(idx=0; idx<size; idx++){
    if(playground[idx]==1) { printf("o"); }
    else                   { printf(" "); }
  }
  printf("\n");
}
```

Configure evolution rules

Initalize ‘all dead’ playground

Draw current playground

Ex. 9: 1D Cellular Automata

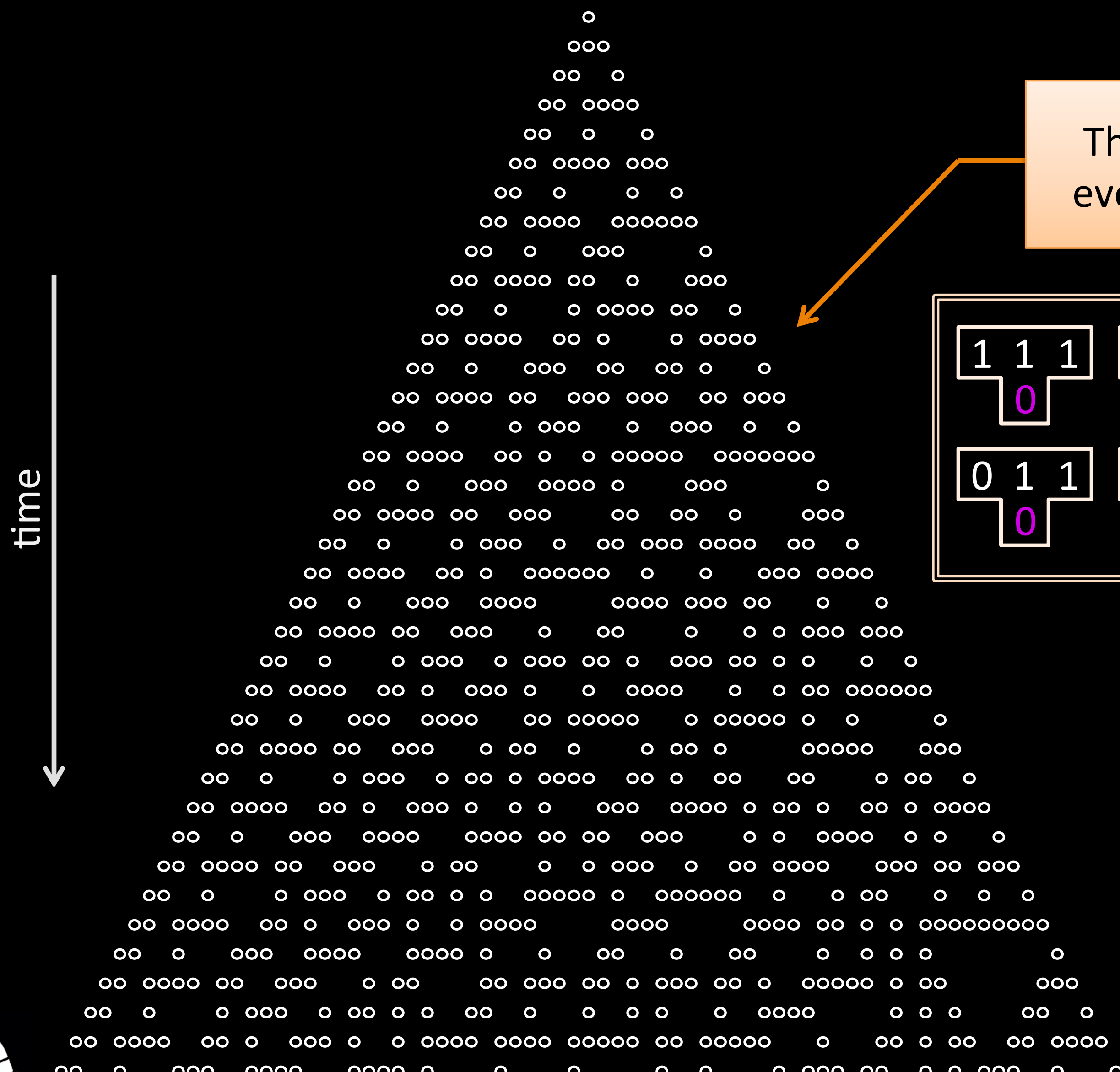
```
void evolve_automaton(int size, int generations, int* evolutionTable, int* playground) {
  int* tmp;
  int idx, cell_idx;
  int c0, c1;
  int* tmp_playground = (int*)malloc(PLAYGROUND_SIZE*sizeof(int));

  print_playground(size, playground);

  for(idx=0; idx<generations; idx++) {
    for(cell_idx=0; cell_idx<size; cell_idx++) {
      c0 = cell_idx==0?size-1:cell_idx-1;
      c1 = playground[c0] * 4 + playground[cell_idx] * 2 + playground[(cell_idx+1)%size];
      tmp_playground[cell_idx] = evolutionTable[c1];
    }

    tmp = tmp_playground;
    tmp_playground = playground;
    playground = tmp;

    print_playground(size, playground);
  }
  free(tmp_playground);
}

int main (int argc, const char* argv[]) {
  int evolutionTable[8];
  int* playground = (int *)malloc(PLAYGROUND_SIZE*sizeof(int));

  init_rules(RULE, evolutionTable);

  init_playground(PLAYGROUND_SIZE, playground);
  evolve_automaton(PLAYGROUND_SIZE, GENERATIONS, evolutionTable, playground);
  free(playground);
  return 0;
}
```

Config. Neighborhood

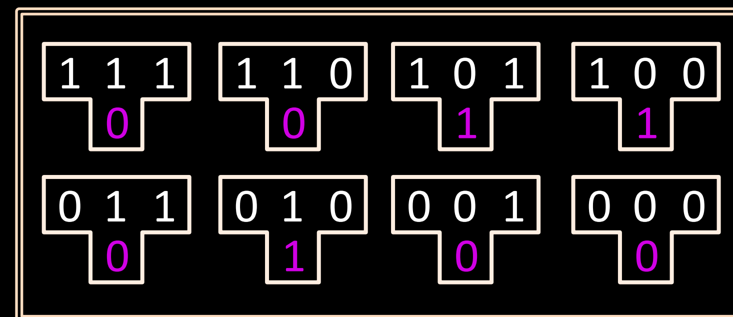
Here's where it happens: Evolution call

Main program: Short, uses functions

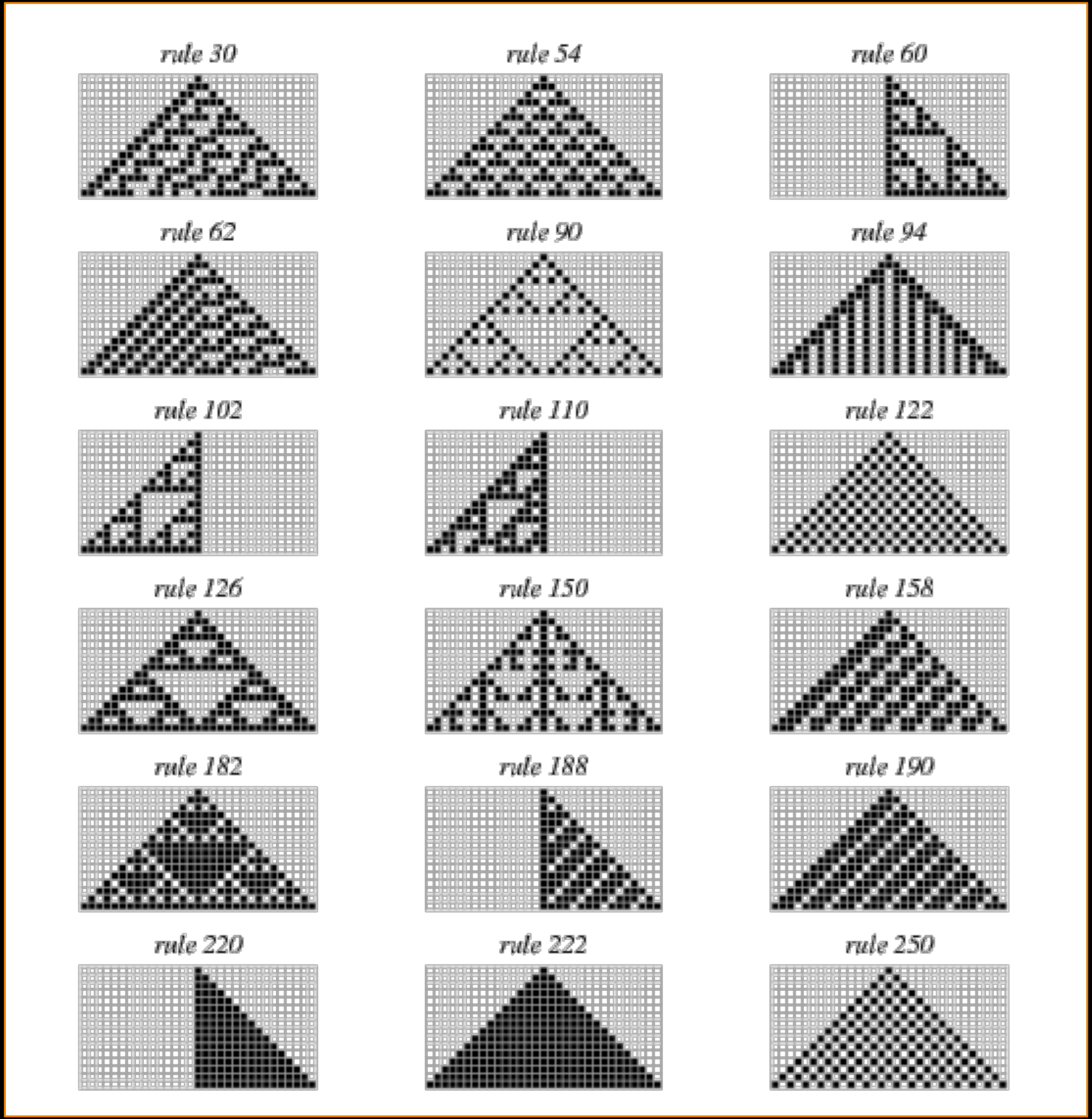
Ex. 9: 1D Cellular Automata



This is an example time evolution using rule 30…



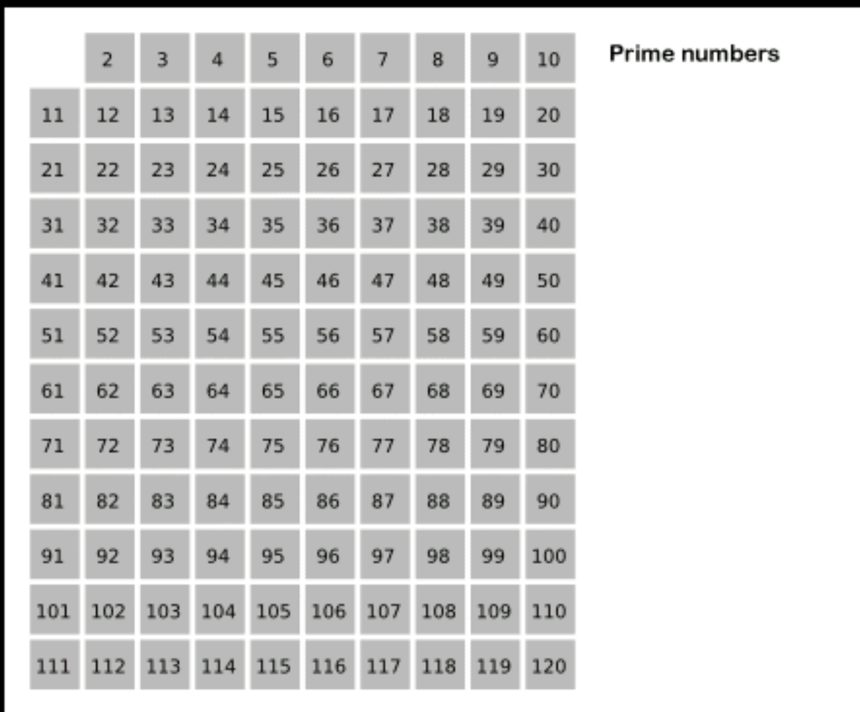
Ex. 9: 1D Cellular Automata



But there are many more!
And they have been extensively studied…

http://mathworld.wolfram.com/ElementaryCellularAutomaton.html

Ex. 10: The Sieve of Erastothenes



(animation from wikipedia)

The Sieve of Eratosthenes is a simple iterative algorithm to generate a table of prime numbers:

- Take the list of the first 100 numbers, and start by removing the multiples of 2.
- Then proceed to remove the multiples of 3.
- Then the multiples of 5 (4 has been removed when we removed the multiples of 2) and so on…
- At the end of this process, what's left are only the prime numbers between 1 and 100.

Write a program to implement this algorithm and use it to calculate the prime factors of an integer number.

See:
http://en.wikipedia.org/wiki/Prime_factor
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Ex. 10: The Sieve of Erastothenes

- There is not much complication in generating the first 100 numbers… But in this solution, let's zero-suppress this array to make things easier:

```
primes[0] = (number of primes)
primes[i] = i-th prime number
```

- Then let's employ these first prime numbers (<100) to determine the prime factors of a large integer number, i.e.

What are the prime factors of **53139008** ?

- N.B.: Since we will just try primes smaller than 100, we won't manage to decompose this number if it has prime factors >100… (but okay, this one will work!)

Ex. 10: The Sieve of Erastothenes

(Let's pretend we don't know this …)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIEVE_SIZE 100
#define NUMBER     53139008 /* 13^2 * 17^3 * 2^6 */
/* #define NUMBER     101 */

int* zero_suppress( int* data, int size){
    int* outcome = (int*)malloc(size);
    //brute force
    outcome[0]=1;
    for(int i=0;i<size;i++){
        if(data[i] != 0 ){
            outcome[outcome[0]++]=data[i];
        }
    }
    return outcome;
}
```

Just a simple function which accepts an array and suppresses zeros. *(and stores size in the first value!)*

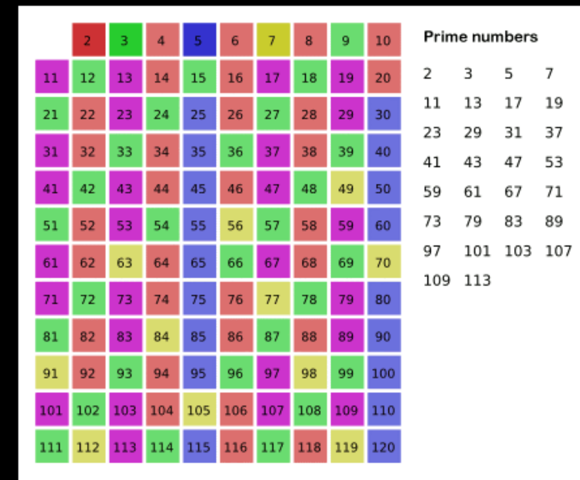
Ex. 10: The Sieve of Erastothenes

```c
/* Brute force prime factors calculation */

int divideAll(int value, int factor) {
    printf("Checking to see if I can still divide with %i (at %i)\n",value,factor);
  if(value%factor==0) return 1+divideAll(value/factor, factor);
  return 0;
}

int check_factorization(int value, int* primes, int* factors) {
  int cvalue, idx;
  for(idx=1; idx<primes[0]; idx++) {
    cvalue *= (int)pow(primes[idx], factors[idx]);
  }
  if(cvalue==value) return 1;
  else              return 0;
}

int* prime_factors(int value, int* primes) {
  int table_size = primes[0];
  int* factors   = (int*)malloc(table_size*sizeof(int));
  int idx;
  for(idx=1; idx<table_size; idx++) {
    factors[idx] = divideAll(value, primes[idx]);
  }

  if(check_factorization(value, primes, factors)) factors[0] = 0;
  else                                            factors[0] = 1;
  return factors;
}
```

Here, we check (recursively!) **how many times** we can divide a certain **`value`** by a **`factor`**

Check if it worked!

Function which calculates the prime factors (indexing as in the sieve)




```
/* The Sieve of Eratosthenes */

int* produce_sieve(int size) {
    int* data = (int*)malloc(size*sizeof(int));
    int idx, done, base;

    for(idx=0; idx<size; idx++) data[idx] = idx;

    done = 0;
    idx = 2;      /* skip 0 and 1 */
    data[1] = 0; /* remove 1 from the sieve */
    do {
      if(data[idx] != 0) {
        for(base=2*data[idx]; base < size; base += data[idx]) data[base] = 0;
      }
      if(++idx>size) { done=1; }
    } while(!done);

    return zero_suppress(data, size);
}
```

This function will take all numbers from 1..99 and set those that are multiples of others to zero (these are not primes!)

Ex. 10: The Sieve of Erastothenes

```
int main (int argc, const char * argv[]) {
  int* primeNumbers;
  int* factors;
  int idx;
  printf("Producing Sieve...\n");
  primeNumbers = produce_sieve(SIEVE_SIZE);
  for(idx=1; idx<primeNumbers[0]; idx++) printf("%d ", primeNumbers[idx]);
  printf("\n");
  printf("Producing factors...\n");
  factors = prime_factors(NUMBER, primeNumbers);
    printf("Produced factors!\n");
  if(!factors[0]) {
    printf("It wasn't possible to fully calculate the prime factors of %d with ", NUMBER);
    printf("a table containing %d primes.\n", primeNumbers[0]);
    printf("This is the best I could do:\n");
    printf("%d ~= ", NUMBER);
  } else {
    printf("%d = ", NUMBER);
  }

  for(idx=1; idx<primeNumbers[0]; idx++) {
    if(factors[idx]!=0) {
      printf("%d^%d ", primeNumbers[idx], factors[idx]);
    }
  }
    printf("\n");

  free(factors);
  return 0;
}
```

(1) Calculate first numbers

(2) Decompose a number

(3) Print numbers

Ex. 10: The Sieve of Erastothenes

Producing Sieve...
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Producing factors...
Checking to see if I can still divide with 53139008 (at 2)
Checking to see if I can still divide with 26569504 (at 2)
Checking to see if I can still divide with 13284752 (at 2)
Checking to see if I can still divide with 6642376 (at 2)
Checking to see if I can still divide with 3321188 (at 2)
Checking to see if I can still divide with 1660594 (at 2)
Checking to see if I can still divide with 830297 (at 2)
Checking to see if I can still divide with 53139008 (at 3)
Checking to see if I can still divide with 53139008 (at 5)
Checking to see if I can still divide with 53139008 (at 7)
Checking to see if I can still divide with 53139008 (at 11)
Checking to see if I can still divide with 53139008 (at 13)
Checking to see if I can still divide with 4087616 (at 13)
Checking to see if I can still divide with 314432 (at 13)
Checking to see if I can still divide with 53139008 (at 17)
Checking to see if I can still divide with 3125824 (at 17)
Checking to see if I can still divide with 183872 (at 17)
Checking to see if I can still divide with 10816 (at 17)
Checking to see if I can still divide with 53139008 (at 19)
Checking to see if I can still divide with 53139008 (at 23)
Checking to see if I can still divide with 53139008 (at 29)
Checking to see if I can still divide with 53139008 (at 31)
Checking to see if I can still divide with 53139008 (at 37)
Checking to see if I can still divide with 53139008 (at 41)
Checking to see if I can still divide with 53139008 (at 43)
Checking to see if I can still divide with 53139008 (at 47)
Checking to see if I can still divide with 53139008 (at 53)
Checking to see if I can still divide with 53139008 (at 59)
Checking to see if I can still divide with 53139008 (at 61)
Checking to see if I can still divide with 53139008 (at 67)
Checking to see if I can still divide with 53139008 (at 71)
Checking to see if I can still divide with 53139008 (at 73)
Checking to see if I can still divide with 53139008 (at 79)
Checking to see if I can still divide with 53139008 (at 83)
Checking to see if I can still divide with 53139008 (at 89)
Checking to see if I can still divide with 53139008 (at 97)

Produced factors!
53139008 = 2^6 13^2 17^3

End result

And that's all, folks

- **In the end, I received only a couple of solutions…**
  - But please take a look! Hopefully (some of it) is useful….
- **Thanks for the effort!**
  - For some, this was too easy, … ("another factorial function…")
  - …for some, maybe too difficult…
  - …but that's fine and expected, as long as we learn something!
- But, if you found it too difficult,
  - Please take another look!
  - Yes, it takes time, but there is no other way
  - If you have questions, **now** is the right time!

Thank you!