Foundations of computation (and mathematics)

- Logic + Set theory (assembly for maths).
- Turing/Von-Neuman machines (imperative).
- Lambda calculus and type theory (FP).
- Category theory (OO/FP).

Why functional?

- **Change of state (=side-effects) are BAD.**
- Functional programming:
  - Functions are expressions.
  - Computation is evaluation of expressions.
  - There are no side-effects.
- Imperative programming:
  - Functions are sequences of statements.
  - Computation is change of state.
  - Has side effects. Can have global side effects.
- OO hides change of state.

Microsoft Excel is a functional programming language!

Some features of functional programming

- Higher order functions.
- Functions have no side-effects (pure functions).
- Referential transparency (consequence of purity).
- Immutable data.
- Lazy evaluation.
- Encapsulation of side effects (monads).
- Recursion (for both types and functions)
- Algebraic Data Types (ADT) and pattern matching.

Functional features in C++

- Has higher order functions (yes/no).
- Functions have no side-effects (yes/no).
- Immutable data (yes/no).
- Lazy evaluation (yes/no).
- Encapsulation of side effects (yes/no).
- Recursion (yes/no).
- ADT (yes/no) and pattern matching (yes/no).

- "yes" on all if you choose to (or try hard).

Functions without side-effects = recursion

```
unsigned int f(unsigned int n) {
    unsigned int res = 1;
    for(unsigned int i = n; i > 0; i--) {
        res *= i;
    }
    return res;
}

unsigned int f(unsigned int n) {
    if(n == 1) return 1;
    else return n*f(n-1);
}
```

Or: be flexible and think: "functions without **global** side-effects".

Functions in C++

- Can be a function.
- Can be pointer to/reference to a function.
- Can be a function objects.
- Can be an object method.
- Can be an object member.
- Can be a lambda expression.
- Can be a partially applied function.
- ...

```
std::function<return type(parameter types)> f = ...
```

Functions which (can) return the square of 10

```
int square10() { return 10*10; }
int square(int n) { return n*n; }
int times(int n, int m) {return n*m; }

struct MySquare {
  int operator() (int n) {return n*n;};
} mySquare;

struct MySquare10 {
  int operator() () {return 10*10;};
} mySquare10;

struct MyNumber {
  int number;
  int times(int m) {return m*number;};
} num;
```

The square of 10 (std::bind)

```
using namespace std::placeholders;
std::function<int(void)> f;       // call f()
std::function<int(int)> g;        // call g(10)
std::function<int(MyNumber&)> h;  // call h(num)
g = square;
f = square10;
f = std::bind(square,10);
g = std::bind(times,_1,_1);
g = mySquare;
f = mySquare10;
f = std::bind(mySquare,10);
num.number = 10*10;
h = &MyNumber::number;
num.number = 10;
f = std::bind(&MyNumber::times,n1,10);
g = std::bind(&MyNumber::times,n1,_1);
```

Lambdas

```
[capture] (parameters) -> return type { body }
```

Continuing...

```
g = [] (int n) {return n*n;};
f = [] () {return 10*10;};
```

Lambdas do not need to be assigned to a function (functions and lambdas are distinct).

```
cout << [] () {return 10*10;}() << endl;
cout << [] (int n) {return n*n;}(10) << endl;
```

Lambdas can be used for initialisation.

```
int m =  [] (int n) {return n*n;}(10);
cout << m << endl;
```

Lambdas

- Parameters are what you would expect.
- Return type is (possibly) deduced by compiler.
- Function variables may use out of scope variables. How?
  - [&] by reference (default)
  - [=] by copy

```
int m = 10;
// m is 10 here
cout << [&p = m, q = m + 1] ()-> int {
                p += 1; return (p-1)*(q-1); }()
        << endl;
// m is 11 here
```

Generic lambdas with auto (no need for templates).

This really makes lambdas shine (since C++14)

```
MyNumber operator*(MyNumber n1, MyNumber n2) {
    MyNumber ret;
    ret.number = n1.number * n2.number;
}

auto i = [] (auto x, auto y) {return x*y;};

cout << i(10,10) << endl;
cout << i(10.0f, 10.0f) << endl;
n1.number = 10;
MyNumber n2; n2.number = 10;
cout << i(n1,n2).number << endl;
```

Lambdas are used for binding

In many ways, std::bind is already obsolete and can be replaced with lambdas.

```
auto f = std::bind(square,10);
auto f = [] () {return square(10);};

auto g = std::bind(times,_1,_1);
auto g = [] (auto x) {return times(x,x);};
```

Overloading ostream

Lets create some a list of integers.

```
vector<int> v(30,0);
vector<int> w(30,0);
list<int> l(30,0);
```

We will need to print them, so we overload ostream. (I might drop std:: and templates due to lack of space)

```
ostream& myout(ostream& os, const T& container) {
    for(auto x : container) os << x << " ";
    return os;
}
ostream& operator<<(ostream& os, const vector<T>& v)
{return myout(os, v);}
ostream& operator<<(ostream& os, const list<T>& v)
{return myout(os, v); }
// other containers inserted here
```

Send me an email if you have a better solution.

Initialising the list with integers 0...29

```
int c = 0;
for(vector<int>::iterator it = w.begin(); it!=w.end(); ++it) {
  *it =  c;
   ++c;
}

int c = 0;
  for(auto& o : l) {
    o = c;
    c++;
  }

for_each(v.begin(), v.end(), [] (auto& x) {
                          static int c = -1; c++; x = c;} );
```

Multiplying each element with 10

```
for(vector<int>::iterator it = w.begin(); it != w.end(); ++it)
  *it = *it*10;

for(auto& ob : v) ob = ob*10;

for_each(l.begin(), l.end(), [] (auto& x) {x = x * 10;});
```

Count the number of elements greater than 100

```
int c = 0;
for(vector<int>::iterator it = w.begin(); it!=w.end(); ++it) {
  if(*it > 100) c ++;
}

int c = 0;
for(auto ob : v) if(ob>100) c++;

for_each(l.begin(), l.end(), [&c] (auto& x) {if(x>100) c++;});

int c = count_if(l.begin(), l.end(),
                 [] (auto x) {return x > 100;});
```

Remove numbers less than a 100

```
std::vector<int>::iterator itf = v.begin();
std::vector<int>::reverse_iterator itb = ++v.rbegin();
while(itf != itb.base()) {
  if(*itf < 100) {
    swap(*itf,*itb.base());
    ++itb;
  } else ++itf;
}
if(*itf >= 100) ++itf;
v.erase(itf,v.end());

auto newlast = remove_if(w.begin(), w.end(),
                     [] (auto x) {return x < 100;});
w.erase(newlast2,w.end());
```

Lets hide some of the complexity

```
auto map = [] (auto p, auto& x) {
  for(auto& ob : v) ob = p(ob));
  };

auto count = (auto p, auto& x) {
  int c = 0;
  for(const auto& ob : v) if(p(ob)) c++;
  return c;
};

auto filter = [] (auto p, auto& x) {
  x.erase(remove_if(x.begin(), x.end(), p),x.end());
  };
```

We initialise, count and remove again.

```
std::vector<int> u(30,0);

map([](auto x){static int c = -1; c++; return c;}, u);

map([](auto x){return 10*x;}, u);

int c = count([](auto x){return x > 100;},u);

filter([](auto x){return x < 100;},u);
```

But still very much mutable.

Lets move to a more functional syntax

```
auto map = [] (auto p, auto& x) {
  for(auto& ob : x) ob = p(ob));
  };

auto count = (auto p, auto& x) {
  int c = 0;
  for(const auto& ob : v) if(p(ob)) c++;
  return c;
};

auto filter = [] (auto p, auto& x) {
  x.erase(remove_if(x.begin(), x.end(), p),x.end());
  };
```

Creating a list of numbers

Lets create $\{x | x = n^2 + 2, n = 1, 2, \cdots, 100\}$

```
vector<int> v(100,0);
map([](auto x){static int c = -1; c++; return c;},v);
map([](auto x){return x*x+2;},v);
```

Is 54 in this list?

We could use STL, but lets not. We could do this:

```
auto linsearch = [] (auto& v, auto val)->bool{
      for(auto& x: v) if(x == val) return true;
      return false;
    };
bool inlist;
inlist = linsearch(v,54); // false
inlist = linsearch(v,18); // true
```

But there are faster ways to search a sorted array.

Is 54 in this list?

```
bool vectorbinsearch(const vector<T>& v, const T& val,
                          int start, int end) {
  if(start > end) return false;
  int i = (start + end) / 2;
  if(v[i] < val)
      return vectorbinsearch(v,val,i+1,end);
  else if(v[i] > val)
      return  vectorbinsearch(v,val,start,i-1);
  else return true;
}

binsearch=[](const auto& v, const auto& val)->bool {
         return vectorbinsearch(v,val,0,v.size()-1);};

inlist = binsearch(v,54);
```

Could overload binsearch, one for each type of container having [] and operators ¡ and ¿. Hm....

Is 54 in this list?

```
// compiler error
auto binsearch = [&binsearch] (stuff) {if(blah) binsearch(stuff)
```

Recursive generic(!) lambdas are a bit cumbersome.

```
auto binsearch = [] (const auto& v, const auto& val)->bool {
  auto h = [] (const auto& v, const auto& val, int start,
                              int end, const auto& f) {
    if(start > end) return false;
    int i = (start + end) / 2;
    if(v[i] < val) return f(v,val,i+1,end,f);
    else if(v[i] > val) return f(v,val,start,i-1,f);
    else return true;
    };
  return h(v,val,0,v.size()-1,h);
  };
```

Send me an email if you can do better.

Is 54 in this list?

What are we using in our binary search?

```
v.operator[]
v.size()
```

and we needed to be able to compare elements with $<$ and $>$.

```
// An array with 10000 ints but no use of memory.
struct MyArray {
    int size() const {return 10000;};
    int operator[](int i) const {return i*i+2;};
} myArray;

inlist = binsearch(myArray,54);  // works fine
```

Can we do something with a list with no data?

```
class MyList {
public:
MyList take(int n) const;
      int operator[](int n) const;
      int length() const;
      MyList map(std::function<int(int)> nf) const;
      MyList filter(std::function<bool(int)> nb) const;
      int head() const;
      MyList tail() const;
      MyList drop(int n) const;
      MyList zipWith(std::function<int(int,int)> p, MyList v);
      MyList concat(MyList v);
      bool infinite() const;
      bool empty() const;
      .....
private:
      // NO DATA
};
```

To make syntax more functional

```
auto take = [] (int n, auto m) {return m.take(n);};
auto filter = [] (auto p, auto m) {return m.filter(p);};
auto map = [=] (auto p, auto m) {return  m.map(p);};
auto head = [] (auto m) {return m.head();};
auto tail = [] (auto m) {return m.tail();};
auto drop = [] (int n, auto m) {return m.drop(n);};
auto length = [] (auto m) {return m.length();};
auto infinite = [] (auto m) {return m.infinite();};
auto empty = [] (auto m) {return m.empty();};
auto zipWith = [] (auto p, auto m1, auto m2)
                     {return m1.zipWith(p,m2);};
auto concat = [] (auto m1, auto m2) {return m1.concat(m2);};
```

Some more details

Private members

```
function<int(int)> f = [] (int x) {return x;};
function<bool(int)> b = [] (int x) {return true;};
...

MyList map(function<int(int)> nf) const {
  MyList ret;
  ret.f = [of=f, lnf=nf] (int x) {
    return lnf(of(x));
    };
  ret.b = b;
  return ret;
  }
```

Sorry about the dense code - was running out of time at this point.

A simple problem

We compute all integers which are relative prime with 100.

```
MyList num;
function<int(int,int)> gcd = [&gcd] (int m, int n)->int {
   if(n==0) return m;
   else return gcd(n,m%n);
   };
num = filter([&gcd] (int x) {return gcd(100,x) != 1;},num1);
cout << take(20,num1) << endl;
```

Sieving primes

We sieve and compute prime number 10001.

```
MyList primes;
primes = drop(2,primes);
auto pf = [] (int n, int x) {return (n != x) && (x % n == 0);};
for(int i = 0; i < 350; i++) {
  int n = primes[i];
  primes = filter([=] (int x) {return pf(n,x);},primes);
}
cout << primes[10000] << endl;
```

(simplified) automatic differentiation

Think of a function f as a pair $(f, \frac{df}{dx})$, which maps a float to a pair of floats. This is almost what we do here:

```
typedef std::pair<float,float> valpair;
typedef std::function<valpair(valpair)> funcpair;
```

Addition, composition and multiplication.

```
auto addpair = [] (funcpair f, funcpair g)->funcpair{
  return [=] (valpair x) {
    return make_pair(f(x).first+g(x).first,
                                f(x).second+g(x).second);
    };
  };

auto mulpair = [] (funcpair f, funcpair g) {
  return [=] (valpair x) {
    return make_pair(f(x).first*g(x).first,
         f(x).first*g(x).second+f(x).second*g(x).first);
    };
  };

auto compair = [] (funcpair f, funcpair g) {
  return [=] (valpair x) {
    return make_pair(f(g(x)).first,
                           f(g(x)).second*g(x).second);
    };
  };
```

That was all we needed

```
funcpair f1 = [](valpair x){
  return make_pair(x.first*x.first*x.first,3*x.first*x.first);};
funcpair f2 = [](valpair x){
        return make_pair(sin(x.first),-cos(x.first));};
funcpair f3 = compair(f2,compair(f1,f2));

cout << f3(make_pair(2,0)) << " " << cos(pow(sin(2),3))*3*
   (sin(2)*sin(2))*cos(2) << " " << sin(pow(sin(2),3)) << endl;
```

Why the pair (2,0)? Well, because (2)'=0. makepair might be obsolete.

Functional programming task

Problem 1. You need to work with lists - I suggest std::vector of ints.
a) Implement the functions

- filter(p,list)
- map(f,list)
- zipWith(f,list1,list2)
- fold(f,list)

Where

- filter removes entries from the list using a function of type bool p(int).
- map applies a function to all elements in a list using a function of type int f(int)
- zipWith(f,list,list) uses a function of type int f(int,int) and returns a list obtained by applying this function to two lists.
- fold(f,list) takes a function of type int f(int,int) and repeatedly applies to the list, using results along the iteration.

Functional programming task

For example

```
If list = [1,2,3,4], p = [] (int x) -> bool {return x < 3;}
then filter(p,list)=[1,2]

If list = [1,2,3,4], f = [] (int x) -> int {return 2*x;}
then map(f,list) = [2,4,6,8]

If list1 = [1,2,3,4] and list2 = [5,6,7,8],
f = []  (int x, int y) -> int {return x + y;}
then zipWith(f,list1,list2) = [6,8,10,12]

If list = [1,2,3,4], f = [] (int x,int y) -> int {return x + y;}
then foldl(f,list) = f(1,f(2,f(3,4))) = 1+2+3+4 = 10
```

Functional programming tasks

b) Implement a function which writes a list to the screen, and a function which initialises a list to contain 1,2,3,...100

c) Then, by only using map, zipWith, etc with pure lambdas (no state-change) solve the following problems.

1. Make a list containing the first 100 odd numbers [1,3,5,...], using map on an initialised list [1,2,3...100])
2. Looping with zipWith and map, make a list containing the sums of odd numbers [1,1+3,1+3+5,1+3+5+7,...]
3. Use map on an initialised list to make the first 100 square numbers [1,4,9,16,...]
4. Use zipWith to make a list which is the difference of elements from the lists in 2 and 3.
5. Compute the sum of the elements in the list from 4 (using fold) and print it to the screen. You will have printed 0 if everything is correct.

Functional programming tasks

Problem 2. Complete the implementation of automatic derivation and test it out on the function $cos(e^{\frac{sin(ln(x^2))}{cos(x)}})$, by drawing the graphs of the automatic differentiated function and the function you differentiate by hand (or Wolphram alpha). Use SFML.
And some **difficult** problems:
Problem 3. Reimplement the lazy-sieve above so that the for-loop is also computed lazily.
Problem 4. Write a wrapper for to std::function so that a function can be saved and read from a file.