# INFORMATIK
# D-MATH / D-PHYS

## Eine Einführung mit C++

Kurzzusammenfassung zur Vorlesung 252-0847-00
HS 2016, ETH Zürich

Felix Friedrich und Bernd Gärtner

# The first program and the C++ syntax and semantics

We defined the scope of the course and that of computer science, followed by a discussion of its essential notion with the oldest handed down example: the Euclidian *algorithm*. We looked at how a computer actually works. We learned to know the important concepts of a Turing Machine and Von Neumann Architecture.

We saw an overview of what it means to program, how to program, and why it is important to be able to program. The main point we made is that basic programming skills are becoming increasingly important in a world "run" by software (programs).

Then we discussed our first C++ program and the main syntactical and semantical terms necessary to understand it.

```
// Program: power8.cpp
// Raise a number to the eighth power.

#include <iostream>

int main()
{
  // input
  std::cout << "Compute a^8 for a =? ";
  int a;
  std::cin >> a;

  // computation
  int b = a * a; // b = a^2
  b = b * b;     // b = a^4

  // output b * b, i.e., a^8
  std::cout << a << "^8 = " << b * b << "\n";
  return 0;
}
```

The two most important syntactical concepts are *expressions* and *statements*. An expression represents a computation (for example `b * b`), and expressions can be combined through operators to new expressions (such as in `b = b * b`). Every expression has a type, a value, possibly an effect, and (if it is an lvalue) an address. Value and effect materialize only in an evaluation of the expression. A statement is the basic building block of every program. It has an effect, and often this effect results from evaluating an expression.

**Learning Goals:** Students should be able to:

- Determine types and values of simple expressions. These are expressions involving

literals or variables of type `int`, `unsigned int`, `bool`, `float`, `double`, `char`, and arithmetic, relational and logical operators (with short-circuit evaluation).

## The types `int`, `unsigned int`, arithmetic operators

Here we introduced the arithmetic operators (on the types `int` and `unsigned int`), along with their precedences (*binding power*), associativities (evaluation from left to right, or right to left?), and arities (number of operands).

| Description | Operator | Arity | Prec. | Assoc. |
|---|---|---|---|---|
| post-increment | `++` | 1 | 17 | left |
| post-decrement | `--` | 1 | 17 | left |
| pre-increment | `++` | 1 | 16 | right |
| pre-decrement | `--` | 1 | 16 | right |
| sign | `+` | 1 | 16 | right |
| sign | `-` | 1 | 16 | right |
| multiplication | `*` | 2 | 14 | left |
| division (integer) | `/` | 2 | 14 | left |
| modulus | `%` | 2 | 14 | left |
| addition | `+` | 2 | 13 | left |
| subtraction | `-` | 2 | 13 | left |
| assignment | `=` | 2 | 3 | right |
| mult assignment | `*=` | 2 | 3 | right |
| div assignment | `/=` | 2 | 3 | right |
| mod assignment | `%=` | 2 | 3 | right |
| add assignment | `+=` | 2 | 3 | right |
| sub assignment | `-=` | 2 | 3 | right |

***Table 1:*** *Arithmetic operators*

Most importantly, the division `/` is the integer division, meaning that `5/2` has value `2`, for example, where the remainder `1` can be obtained through the modulus operator (`5%2`). The three main rules one needs to remember are:

**Arithmetic Evaluation Rule 1:** Multiplicative operators have higher precedence than additive operators.

**Arithmetic Evaluation Rule 2:** Binary arithmetic operators are left associative.

**Arithmetic Evaluation Rule 3:** Unary operators `+` and `-` have higher precedence than their binary counterparts.

We have also talked about the value ranges of the types `int` and `unsigned int`. Under a $b$-bit representation (think of $b = 32$), `int` has value range $\{-2^{b-1}, -2^{b-1}+1, \ldots, -1, 0, 1, \ldots, 2^{b-1}-1\} \subset \mathbb{Z}$, while `unsigned int` has value range $\{0, 1, \ldots, 2^b - 1\}$.

**Learning Goals:** Students should be able to:

- convert natural numbers from and to decimal, binary and hexadecimal representation.

# The type `bool`

We have introduced the type `bool` to work with truth values, along with the logical operators (that combine boolean expressions), and the relational operators (that compare arithmetic expressions and return boolean expressions). A specialty of boolean expressions

| Description | Operator | Arity | Prec. | Assoc. |
|---|---|---|---|---|
| logical not | `!` | 1 | 16 | right |
| less | `<` | 2 | 11 | left |
| greater | `>` | 2 | 11 | left |
| less or equal | `<=` | 2 | 11 | left |
| greater or equal | `>=` | 2 | 11 | left |
| equality | `==` | 2 | 10 | left |
| inequality | `!=` | 2 | 10 | left |
| logical and | `&&` | 2 | 6 | left |
| logical or | `\|\|` | 2 | 5 | left |

***Table 2:*** *Logical and relational operators*

built from `&&` or `||` is the short circuit evaluation: if the value is already known after evaluation of the left operand, the right operand is not evaluated. This allows for expressions such as `x == 0 || y / x > 2` where the second operand is valid only if $x \neq 0$.

We learned to know the important concept of *assertions* to detect errors in a program. An assertion can be used to formulate invariants or preconditions in a program. We discussed differences between compile-time and run-time errors.

# Control statements

The major part of this lecture was about control statements that allow us to deviate from linear control flow (statements are executed from top to bottom in the program). The control statements are

- the `if`- and the `if-else` statement (*selection* statements);

```
if ( condition )
     statement
else
     statement
```

- the `for`-statement (*iteration* statement); this is at the same time the most used, the most important, and most complex control statement;

```
for ( init statement condition ; expression )
     statement
```

- the `while` and the `do`-statements (two more *iteration* statements); since they can both easily be simulated using the `for`-statement, they are redundant from a functional point of view, so we called them *syntactic sugar*. However, they still make sense and allow for more readable code;

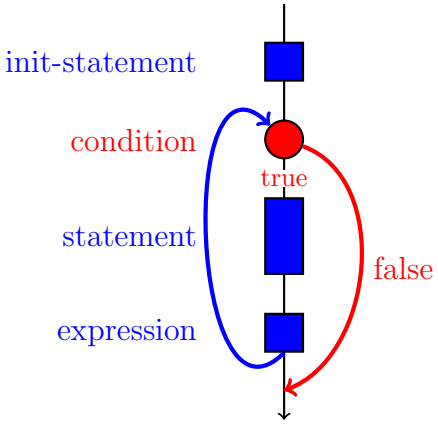
```
while ( condition )
     statement
```

- `break;` and `continue;` (*jump* statements). The typical use of `break;` is to terminate a loop in the middle, and the use of `continue;` is to skip the remainder of a loop iteration.

In this context, we discussed undecidability of the *halting problem*: there is no C++ program that can check any C++ programs for termination.

We also discussed *blocks* as a means to group several statements into one, and we said that each block is a *scope* in the sense that variables defined in the block start to live (are accessible and get memory assigned) when the program control enters the block and die (get their memory freed and become inaccessible) when the block is left. This is the concept of *automatic storage duration*.

We looked at the control-flow of the different statements and learned how they help to avoid the production of Spaghetti code via uncontrolled branches.



**Learning Goals:** Students should be able to:

- Determine the behavior of simple code. These are complete programs or sequenes of statements containing simple expressions, in- and output, selection statements (if, if-else) and iterations statements (for, while, do)
- Write simple code for given tasks.

# Floating point numbers

We have discussed how to compute with "real" numbers in C++ using floating point numbers and the types `float` and `double`. Floating point numbers are numbers of the form $\pm s \cdot 2^e$, where both $s$ (the significand) and $e$ (the exponent) are integers. Concrete floating point number systems (such as *IEEE single precision* and *IEEE double precision*) prescribe bounds for $s$ and $e$.

We have seen that floating point numbers are very useful for a wide range of applications (we have computed the Euler number and Harmonic numbers in the lecture), but that they come with some pitfalls that one needs to know. Most prominently, we discussed that floating point inputs and literals are usually decimal (such as `0.1`, meaning $1/10$), but that internally, a binary representation is used. In converting between these two representations, errors are unavoidable, since numbers with finite decimal representation may have infinite binary representation. This can lead to small errors (0.1 internally not being 0.1 but very close to it), and large errors (Excel 2007 bug).

We established three guidelines for safely computing with floating point numbers.

**Floating Point Arithmetic Guideline 1:** Never compare two floating point numbers for equality, if at least one of them results from inexact floating point computations.

**Floating Point Arithmetic Guideline 2:** Avoid adding two numbers that considerably differ in size.

**Floating Point Arithmetic Guideline 3:** Avoid subtracting two numbers of almost equal size, if these numbers are results of other floating point computations.

What happens when we violate the second one was demonstrated by a program for computing Harmonic numbers in two ways (forward and backward); only the backward

way is approximately correct, while in the forward way, the numbers `1/i` added last are too small to still contribute.

```
// Forward sum, yields large errors for large n
float fs = 0;
for (unsigned int i = 1; i <= n; ++i)
  fs += 1.0f / i;

// Backward sum, approximately correct also for large n
float bs = 0;
for (unsigned int i = n; i >= 1; --i)
  bs += 1.0f / i;
```

**Learning Goals:** Students should be able to:

- Work with floating point number systems. In particular be able to convert numbers between decimal and binary floating point number systems.
- Understand the limits of floating point numbers.

# Functions

We have discussed functions as a way to realize functionality that is frequently used (for example, power computations) in such a way that it is implemented only once (as a function) but can be used very often, and even from different programs. We have seen that the standard library contains many of such functions (e.g. `std::sqrt`).

We have made the point that functions should be properly documented through pre- and postconditions, and that assertions should be used to check pre- but also other conditions at any time during a program.

We have also seen that function calls are expressions, and how the evaluation of a function call proceeds (evaluation of call arguments, initialization of formal arguments, execution of function body, return of function value (optional for `void` functions)).

We have learned how to structure a problem into manageable blocks by using the concept of *stepwise refinement* applied to *procedural programming*. We used the example of intersecting rectangles.

---

```
#include<iostream>
#include<algorithm>
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
// with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect (int a1, int b1, int a2, int b2) {
    return std::max(a1, b1) >= std::min(a2, b2)
```

```
            && std::min(a1, b1) <= std::max(a2, b2);
}
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
// w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
bool rectangles_intersect (int x1, int y1, int w1, int h1,
int x2, int y2, int w2, int h2){
    return intervals_intersect (x1, x1 + w1, x2, x2 + w2)
            && intervals_intersect (y1, y1 + h1, y2, y2 + h2);
}
int main (){
    ...
    bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);
    ...
    return 0;
}
```

We learned to know the concept of separate compilation: components of a program can be compiled independent of each other provided that their interface is given in some header-file. In order to avoid name-clashes in large libraries we discussed the notion of namespaces.

**Learning Goals:** Students should be able to:

- Find pre- and postcondions of functions and evaluate function calls
- Understand and write simple functions for given tasks. A simple function is a function whose body consists of simple code, and which provides call-by-value arguments.

# Reference types

In order to motivate reference types we looked at a simple but often required functionality: swap the values of two variables. A first trial to write a swap function without reference parameters failed. We learned that the application of reference types at the parameters of a function provides a solution.

```
// POST: values of a and b are exchanged
void swap (int& a, & b) {
 int t = a;
 a = b;
 b = t;
}
```

```
int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a==1 && b == 2);
}
```

As another concept that helps to write more failsafe code we learned to know the concept of constants. The compiler guarantees that any variable declared as constant will not be modified. We learned a rule for writing const-correct programs:

> **Const Guideline:** Whenever you define a variable in a program, think about whether its value is supposed to change or not. In the latter case, use the const keyword to turn the variable into a constant.

If a formal argument is of reference type, we have *call by reference*, otherwise *call by value*. For large types, call by value can be inefficient, since the whole value is copied. We can avoid this by using *const references* instead and are still able to pass rvalues.

We learned that even return by reference is possible, a concept that gave rise to the following guideline.

> **Reference Guideline:** When you define a reference, the referred to object must be live at least as long as the reference itself.

**Learning Goals:** Students should be able to:

- Understand and write simple code and simple functions that deal with functions that call other functions and that use call-by-reference.

# Arrays and vectors

We have seen the concept of an array as a container that stores a sequence of data of the same type, and allows fast random access (by index) to the elements. This is possible since an array occupies a consecutive piece of memory, so the address of any element is easy to compute as $\mathtt{p} + \mathtt{si}$, where $\mathtt{p}$ is the address of the first element, $\mathtt{i}$ is the index of the element, and $\mathtt{s}$ is the number of memory cells required to store a single element.

We have seen two models of the array concept: static arrays (the number of elements has to be known at compile time) and vectors (the number of elements can be determined at runtime). Static arrays turn out to be primitive (they originate from C), in the sense

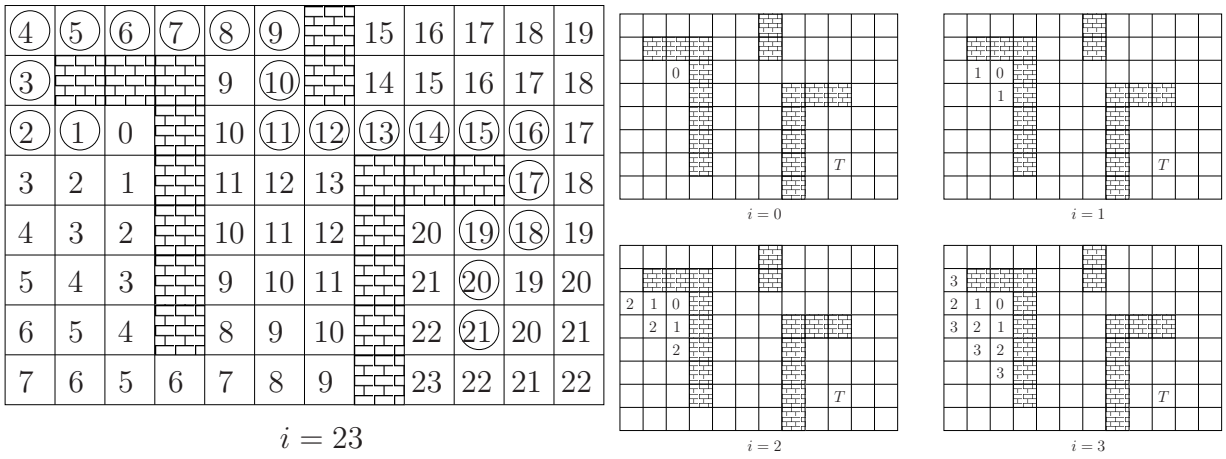
that initialization and assignment don't work as for other types, and we have to "copy arrays by hand".

As applications for arrays, we have considered (i) Eratosthenes' sieve for computing all prime number up to a given upper bound, and (ii) Cesary-code encoding and decoding: replace all printable characters in a text by their predecessor and successor, respectively.

For the latter, we also needed the type `char` to represent single (ASCII) characters.

We introduced strings and operations on strings with the very appealing example of Lindenmayer-systems. We learned to know an alphabet and production rules on the set of finite words over the alphabet and how to implement such rules in a program.

Then we have discussed multidimensional arrays (arrays of arrays of arrays...), with one (twodimensional) application: finding shortest paths for a robot on a factory floor from a start to a goal position that avoids all obstacles.




The main idea in the solution was to label every cell with the length of the shortest path from the start to the cell (a twodimensional array stores these labels), and use an expanding wavefront to assign labels in increasing order. In the left picture, the cell labeled `0` is the start, and the cell labeled `21` is the goal. The indicated path of increasing labels is the shortest path. A single step of expanding the wavefront to label `i` is really simple (right picture): all unlabeled neighbors of cells with label `i - 1` receive label `i`.

**Learning Goals:** Students should be able to:

- Understand and write simple code and simple functions that deal with one- and twodimensional arrays and vectors

# Pointers, Algorithms, Iterators, Containers

We have discussed how to pass arrays to functions: via two *pointers*, one pointing to the first element, and one *behind* the last element. A pointer of type `T*` stores the address

of an object of type `T`. A static array is automatically converted to a pointer to its first element in every expression, and using *pointer arithmetic*, we can get pointers to other elements. For example, to set all 5 elements of an array `a` to 1, we can use

```
// a points to first element, a+5 behind the last
std::fill (a, a+5, 1);
```

Here, `std::fill` is an *algorithm* from the standard library. In realizing such a function for an array of length `n`, we have discussed iteration by random access, and natural iteration:

```
// iteration by random access: read book
// constantly opening and closing it between pages
for (int i=0; i<n; ++i)
   a[i] = 1;

// natural iteration: read book
// opening it once and turning pages
for (int* p = a; p < a+n; ++p)
   *p = 1;
```

For vectors, we cannot use pointers, but have to work with *vector iterators* that can be thought of as "pointers" to vector elements. To set all elements of a vector, we use

```
// v.begin() points to first element, v.end() behind the last
std::fill (v.begin(), v.end(), 1);
```

An iterator is a type whose expressions behaves like pointers, meaning that an expression `it` of iterator type can at least be dereferenced to yield the element it points to (`*it`), and advanced to point to the next element (`++it`). Some iterators (for example, pointers and vector iterators) also support random access (`it[k]`).

A *container* is an object that can store other objects (its elements), and that offers some ways of accessing these elements. Every standard container allows access through iterators. If `c` is a standard container of type `C`, the following loop will output all its elements.

```
for (C::const_iterator it = c.begin(); it != c.end(); ++it)
   std::cout << *it;
```

Concretely, we have seen this loop for `C` equal to `std::set<char>`, a standard container to represent sets of characters. Set iterators do not allow random access, since from a mathematical point of view, a set is not ordered: there is no natural way to define the `i`-th element.

**Learning Goals:** Students should be able to:

- Understand and write simple code and simple functions that deal with pointers, iterators and ranges.

# Recursion

We have discussed *recursive* functions, functions that call themselves within their body. Here is an example for computing the Fibonacci numbers and the direct counterpart in C++

$$
\begin{aligned}
F_0 &:= 0, \\
F_1 &:= 1, \\
F_n &:= F_{n-1} + F_{n-2}, \quad n > 1.
\end{aligned}
$$

```
// POST: return value is
// the n-th Fibonacci number F_n
unsigned int fib (const unsigned int n)
{
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2); // n > 1
}
```

For a recursive function, we separately have to prove *correctness* and *termination*.

The evaluation of a recursive function call may take very long (Example: `fib(50)`). In many cases there are efficient iterative (non-recursive) ways of writing the same function.

We have discussed the *call stack* as a way to deal with several formal arguments that are in use simultaneously during the evaluation of nested function calls.

We have introduced a major example for the rest of the semester: the command line calculator. We investigated the structure of expressions and their evaluation. We used the Extended Backus Naur Form (EBNF) to reveal the inherently recursive nature of expression parsing.

```
factor = unsigned_number | "(" expression ")" | "" factor.
term = factor { "*" factor | "/" factor }.
expression = term { "+" term | "-" term }.
```

Having developed this form, we could very naturally, and much simpler than in the iterative case, develop parsing for a command line calculator, by almost literally translating the EBNF to functions.

```
double term (std::istream& is)
{
  double value = factor (is);          // factor
  while (true) {
    if (consume (is, '*'))
      value *= factor (is);            // "*" factor
    else if (consume (is, '/'))
      value /= factor (is);            // "/" factor
    else
      return value;
  }
}
```

EBNF only defines the syntax and does not talk about the meaning (value) of an expression. For example, the left-to-right evaluation order within a product of factors or a sum of terms is not encoded in the expression's syntax. In fact, starting with the natural BNF variant of the very same grammar, the literal translation into parser code leads to a right-to-left evaluation order.

**Learning Goals:** Students should be able to:

- Understand and write simple code and simple functions that deal with recursion.
- Work with formal grammars such as Linednmayer systems and EBNF.

## Structs and Classes

We have seen *structs* as a way to define new types whose value range is the Cartesian product of existing types. Our running example was that of rational numbers:

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

We have seen how to provide functionality for a new type (such as `rational`), in order to make it behave like existing types. We learned that C++ supports the concept of function overloading that, once knowning that operators can be defined like functions, can be used to support *operator overloading*.

For example, to enable us to write `r + s` if `r` and `s` are of type rational, we overload the binary addition operator:

```
// POST: return value is the sum of a and b
rational operator+ (const rational a, const rational b)
{
  rational result;
  result.n = a.n * b.d + a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

We can do the same with all other arithmetic operators, relational operators, input and output operators. Here the previously introduced concept of reference types proved to be essential for the correct implementation of overloaded operators such as `+=`.

**Learning Goals:** Students should be able to:

- Understand and write simple code, simple functions and overloaded operators that in addition deal with simple structs. A simple struct is a struct without member functions.

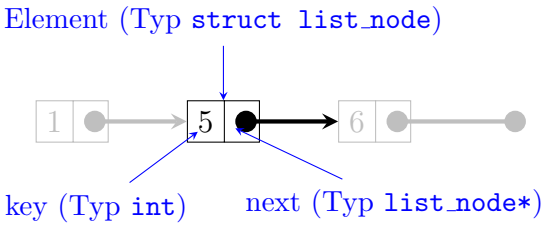
# Classes

We have addressed the main drawback of structs so far, namely that the internal representation (`.n` and `.d` for rational numbers) is revealed to the user ("customer") of the struct, meaning that it is hard to change it later. With classes, we hide the internal representation as it is *private*. Access is provided via member-functions in the public part.

```
class rational {
public:
    // POST: return value is the numerator of *this
    int numerator () const {
        return n;
    }
    // POST: return value is the denominator of *this
    int denominator () const {
        return d;
    }
private:
    int n;
    int d; // INV: d!= 0
};
```

With classes, we hide the internal representation and allow access to it only through *public member functions*. This means, we promise a certain functionality, but not a concrete realization of it. We exemplified this with the running example of rational numbers.

Having learned about encapsulation with classes and having introduced constructors, we discussed the implementation of a class for a stack. We have seen that for arrays, it is difficult to insert or remove elements "in the middle", and we have proposed *linked lists* as a solution that may store their elements anywhere in memory (and not consecutively as arrays do it). In addition to the *key* (or value), each list element also stores a pointer to its successor:




```
struct list_node {
  int             key;
  list_node*      next;
  // constructor
  list_node (int k, list_node* n)
    : key (k), next (n) {}
};
```

The main new concept we needed in order to insert new elements was *dynamic memory allocation* from the heap using `new` (and a later `delete` to free the memory again). We learned the memory allocation rule:

> Allocation (`new`) and de-allocation (`delete`) are conceptually always provided as pairs.

Allocation was, for example, used when pushing elements on the stack

```
void stack::push (int value) {
    top_node = new list_node (value, top_node);
}
```

Deallocation was used when removing elements from the stack

```
void stack::pop() {
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

We learned another important technique to decouple the high level interface of a particular data type such as a stack from the relatively low-level implemenation as linked list: provision of a container class (`stack`) that hides the implementation details of the data structure (`list_node`) and that provides the high-level interface to initialization, allocation and deallocation of the respective data type.

In order to override default initialization and assignment of a stack that would just copy the pointer to the first element (making the new stack an alias of the old one), we needed to provide user-defined *copy constructor* and *assignment operator*. To properly clean up the stack (`delete`!) when it runs out of scope, we also need to provide a user-defined *destructor*, calling a clear method.

```
stack::~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

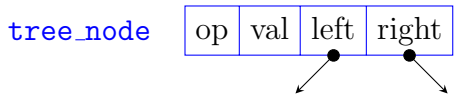
Having constructors, destructor, copy constructor and assignment operator (recall the C++*rule of three*), our type is a *dynamic data type*.

**Learning Goals:** Students should be able to:

- Simple classes. A simple class is a class with member functions, but without dynamic memory management:
- General classes, including dynamic memory management.

# Inheritance and Polymorphism

As a motivating example for this part, we looked at expression *trees* as a natural generalization of linked lists with two successor nodes:




```
struct tree_node {
  char op;
  // internal node ( op: '+, '-', '*', '/')
  tree_node* left;
  tree_node* right;
  // leaf node (op: '=')
  double val;
  // constructor
  tree_node (char o, tree_node* l, tree_node* r, double v)
    : op (o), left (l), right (r), val (v)
  {}
};
```

We implemented various operations on expression trees (such as `eval`, `size`, `copy` and `clear`) by recursive tree traversal along the left and right edges of each tree node.

In complete accordance with the stack example, we implemented a container class `texpression` for expression trees. Moreover, we defined operators on expression trees that allowed it to use trees in the same way as we used numbers before. This went so far that

we could even replace the number type `double` of an expression parser by an expression tree node.

```
texpression operator  (const texpression& l,
const texpression& r)
{
    texpression result = l;
    return result    = r;
}

expression a = 3;
texpression b = 4;
texpression c = 5;
texpression d =  a   b   c ;
```

We motivated the two remaining principles for *object oriented programming* with the wish to extend the expression tree without major modification of existing code.[1]

We learned that classes can inherit properties from other classes and that code written for the *base class* can be reused by the *inheriting class*.

```
struct xtree_node{
    virtual int size() const;
    virtual double eval () const;
};

struct number_node : public xtree_node {
    double val;
    int size () const;
    double eval () const;
};
```

By *overwriting* member functions, inheriting classes can also modify behavior. We learned that, if a member function is labeled `virtual`, the *dynamic* type determines the member function to be executed during runtime. This allows *polymorhpy*, i.e. the fact that the same symbol can adopt different behavior depending on its runtime type.

[1] Inheritance and Polymorphism will not be relevant for the exam.