Informatik

Fynn Krebser–fkrebser@student.ethz.ch

Herbstsemester 2025

Contents

1 Introduction

1.1 Computer Science

Computer Science used to be part of mathematics and was practiced before the invention of computers. Today, it is its own discipline. It can be summarized as

Definition 1.1: Computer Science

COMPUTER SCIENCE is concerned with the automation of intellectual activities.

In this course one will learn how to write a computer program and what happens when a computer executes one.

1.2 Algorithms

Alogrithms form the core of computer science.

Definition 1.2: Algorithm

An ALGORITHM is

- A series of instructions to solve a problem step by step.
- Its execution does not require any intelligence, but precision.

A simple example of an algorithm is the recipe for a cake. However, algorithms for computers need to be even simpler as computers are pretty stupid.

The oldest non-trivial algorithm is the Euclidean algorithm to compute the greatest common divisor (gcd) of two integers a and b.

Algorithm 1.3: Euclidean Algorithm

The Euclidean algorithm works as follows:

- We are given two integers a and b.
- If $a = b$, we have found the gcd, the algorithm terminates.
- W.l.o.g. assume $a > b$. We replace a by $a - b$. The algorithm restarts with the values $a-b > 0$ and b.

Firstly notice that this algorithm terminates as either a or b is getting smaller in each step. Secondly, we shall prove that this algorithm indeed computes the gcd of a and b.

Proof. [Proof of Euclidean Algorithm] Assume W.l.o.g. that $a > b$. We define $d = \gcd(a, b)$. Clearly $d|a \wedge d|b \Rightarrow d|(a - b)$. It is easy to see that if we have $a = b$ then they are also equal to d. □

When implementing an algorithm, there are three LEVELS OF ABSTRACTION one goes through:

- Core idea (abstract)
- Pseudocode (semi-detailed)
- Implementation (very detailed)

So far, we have seen the core idea of the Euclidean algorithm. A possible pseudocode implementation is as follows:

In order to implement we postulate a box-computer which comprises a set of boxes labeled with ascending integer numbers. We call the labels ADRESSES and the boxes MEMORY LOCATIONS which can contain data or instructions.

Instructions are simple statements like reading or writing data to memory locations, arithmetic operations or jumps to other instructions etc.

The euclidean algorithm can be implemented on such a box computer as depicted in figure 1.

Algorithm 1: Pseudocode for the Euclidean Algorithm

Input: Two integers a, b
Output: The gcd of a and b
while $a \neq b$ **do**
  if $a > b$ **then**
    $a \leftarrow a - b$;
  else
    $b \leftarrow b - a$;
  end
end
return a;

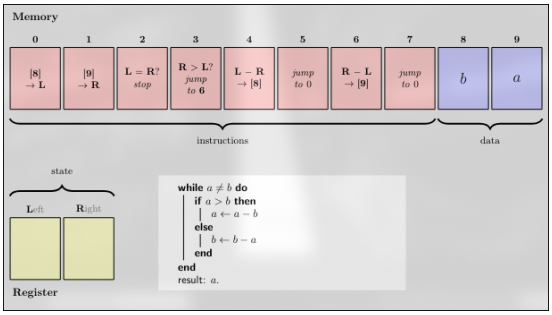

Figure 1: A simple box computer performing the euclidean algorithm.

1.3 Computers

Alan Turing was one of the first to formalize the concept of a computer. He invented the Turing machine which is a theoretical model of a computer which is capable of performing any mathematical computation if it was represented as an algorithm. A machine is called TURING COMPLETE if it can simulate a Turing machine.

Definition 1.4: Turing Machine

A Turing machine consists of an infinitely long tape that can store symbols. There is a read/write head, which can be moved along the tape. Controlling the head is done by a pre-programmed computer that can interpret symbols as instructions and can perform very simple operations on its internal state.

While the first computers were extremly slow todays computers can perform approximately 200 - 800 billion INSTUCTIONS PER SECOND (IPS).

1.4 Programming

As we have seen in the box computer example, a computer needs extremely basic intructions, which even vary from computer to computer. To solve the problem of communicationg with a computer, we use programming language which are then translated into machine code.

Like a standard language, a programming language contains a syntax and semantics. Syntax tells us wheter a text can be considered a valid program. A compiler can tell us if a program is syntactically correct. Semantics however tells us what a program does. This usually requires human understanding.

2 A C++ Program and its Components

We analyze the following C++ program:

```
1  // Program: power8.cpp
2  // Raise a number to the eighth power.
3  #include <iostream>
4  int main() {
5      // input
6      std::cout << "Compute a^8 for a = ? ";
7      int a;
8      std::cin >> a;
9      // computation
10     int b = a * a; // b = a^2
11     b = b * b; // b = a^4
12     // output b * b, i.e., a^8
13     std::cout << a << "^8 = " << b * b << "\n";
14     return 0;
15 }
```

Using the program above, we can identify several components of a C++ program.

2.1 Structure of a C++ Program

Every C++ consists of a MAIN FUNCTION which is the entry point of the program.

Lines starting with `//` are comments and are ignored by the compiler. They are used to explain the code to humans.

The heart of the program is the `main` function. It contains several STATEMENTS which are executed sequentially. Each statement has to end with a SEMICOLON which is the statement terminator in C++. It is good practice to write one statement per line.

2.2 Output

A first statement in our program can be found on line 6. For now, we don't need to understand the details of this statement of `std::cout «`, just that what comes after it is printed to the screen.

If we don't have the first line of the program, the program would result in a compiler error, as the first line includes the `iostream` library which contains the definition of `std::cout`. For simplicity, we will often ignore the `#include` statements in the following sections.

2.3 Variables: Declaration, Initialization, and Assignment

When working with variables, we first DECLARE them.

Definition 2.1: Variable Declaration

A VARIABLE DECLARATION introduces a new variable with a name and a type. The type determines what kind of values the variable can hold.

In C++, a variable declaration has the form `<type> <name>;`.

```
int a; // declares an integer variable named a
```

Typical types are `int` for integers, `unsigned int` for natural numbers or `float` and `double`. Variables can be named feely as a sequence of letters, digits, and underscores, but typically starting with a lower-case letter. for floating point numbers.

We can store a value in a variable using an ASSIGNMENT. This is done by the ASSIGNMENT OPERATOR `=`. For example, `a = 5;`.

Notice that if we assign a variable a value on the same line as we declare it, we call this INITIALIZATION. For example, `int a = 5;`.

Instead of assigning a value to a variable, we can also assign it the value of an expression, which is first computated and then assigned to the variable.

To read or access a variable, we simply use its name. For example, `std::cout « a;` prints the value of the variable `a` to the console.

2.3.1 Pitfalls with Variables

The proably most common mistake when working with variables is to have incompatible types. What happens when we try to assign a floating point number to an integer variable for example? In many cases, C++ performs an IMPLICIT CONVERSION, for example by turning a floating point number into an integer by dropping the decimal part (3.7 becomes 3). As one can imagine, this can lead to unexpected results.

> Definition 2.2: **Compatible Types**
>
> Two types are called COMPATIBLE if C++ can perform an implicit conversion from one type to the other.
>
> Otherwise, the types are called INCOMPATIBLE.

Using two incompatible types in an assignment results in a compiler error.

Another common mistakes is to confuse the assignment operator `=` with the comparison operator `==` which tests for equality. In C++, `a = b` corresponds to the mathematical expression $a := b$.

This leads to the possibility to write statements such as `x=x+1;` which would be mathematical nonsense.

Another difference to mathematics is that statements such as `2 = x;` would result in a compiler error as the left hand side of an assignment has to be a variable. Formally, we use the term LVALUE for expressions that can appear on the left hand side and RVALUE for expressions that can appear on the right hand side. The latter do not contain an adress in memory and thus cannot be assigned to.

Lastly, always be careful to initialize variables before using them. Otherwise, they contain garbage values which will most likely lead to unexpected results.

2.4 Input

So far, we could make our program always compute a^8 for a fixed value of a. To make our program more useful, we want to read the value of a from the user. This is done using the `std::cin` functionality. The statement `std::cin » a;` reads a value from the console and stores it in the variable `a`. Note that the variable `a` has to be declared before it can be used.

2.5 Expressions, Operations and Expression Statements

In C++ an EXPRESSION is quite general.

> Definition 2.3: **Expression**
>
> An EXPRESSION represents a computation in C++. It has a fixed *type*, a *value*, and potentially an *effect*.

While the type is determined at compile time, the value and effect are only known at runtime.

We call an expression PRIMARY if it is just a variable or a literal value such as `5` or `3.14`, otherwise it is a COMPOSED expression.

If an expression is just written on a single line and ends with a semicolon, we call this an EXPRESSION STATEMENT. For example, `a + 5;` is an expression statement, as it does not get any further usage. This seems useless, unless our potential effect, also called SIDE EFFECT, happens to do something useful. For example, the assignment expression `a = 5;` has the side effect of changing the value of `a`. This makes it possible to write statements such as `std::cout « a = 5;` which first assigns the value 5 to `a` and then prints it to the console.

Other examples of operators with side effects are `«` and `»`, which we have already seen in the context of input and output.

2.6 Constants

As seen, variables can change their value during the execution of a program. Sometimes, we want to prevent this for example the speed of light should never change. This can be done using CONSTANTS.

> Definition 2.4: **Constant**
>
> A CONSTANT is a variable, whose value cannot be altered after its initialization. In C++, constants are declared using the keyword `const`.
>
> ```
> const double c = 299792458; // speed of light
> ```

Attempting to change the value of a constant results in a compiler error. In general, it is good practice to use constants whenever possible. A program that uses contants wherever possible is called CONST-CORRECT.

3 Expressions

We have seen what an expression is, now we will look at the semantics of expressions.

3.1 Arithmetic Expressions

When playing around with arithmetic expressions, we notice that C++ follows the standard order of operations. This may be surprising, if we compare it with a cheap calculator, which usually evaluates expressions from left to right. We consider the following operation:

```
int farenheit = 32 + 9 * celcius /5;
```

This expression contains three LITERALS (32, 9, 5), one VARIABLE (celcius) and three OPERATOR SYMBOLS (+, *, /). The order of operations comes from PRECEDENCE AND ASSOCIATIVITY.

3.2 Precedence and Associativity

As in math certain operators have priority over others.

> **Definition 3.1: Precedence**
>
> Precedence is the order in which operators are evaluated. Operators with higher precedence are evaluated before operators with lower precedence. For example, multiplicative operators (*, /, %) have higher precedence than additive operators (+, -).

Similarly, there is a so called LEFT ASSOCIATIVITY for operators with the same precedence. This means that operators are evaluated from left to right.

Another rule is that if for example a minus sign is used as a unary operator like in -3-4 (opposed to -(3-4)), it has a higher precedence than the binary minus operator.

> **Definition 3.2: Arity**
>
> Unary operators +, - have higher precedence than the binary operators.

Last but not least, parentheses have the highest precedence and can be used to override the default precedence and associativity rules.

3.3 Expression Trees and Evaluation Order

An expression can be represented in a tree, connecting two operands by a operator node. The above example for example would be seen in figure 2.

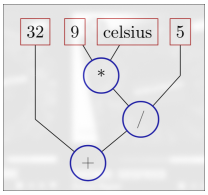

Figure 2: Expression Tree for 32 + 9 * celcius / 5

Usually in computer science, this tree would be displayed upside down, with the root at the top. An expression in C++ is VALID, if every node is evaluated after its children. As a guideline from this: *Avoid modifying variable that are used in the same expression more than once.*

3.4 Arithmetic Operators

The precedence and associativity of arithmetic operators is well defined in C++ and can be found in the reference. We will now look at some special cases, starting with RIGHT-ASSOCIATIVE operators.

An example of this is the Assignment operator (=). This can be useful for multiple assignement like `a = b = 0;`

Another interesting operator is the DIVISION OPERATOR (/). For now, let's look at the integer division. For positive numbers it delivers the result of the division without the remainder. For example `5 / 2` yields 2. One should always be cautious resulting in the following guideline:

Watch out for potential loss of precision and as a consequence postpone operation that may lead to loss of precision as long as possible.

The modulo operator (%) also has some special properties. Mainly, that for negative numbers, the result will also be negative. To fix this one could use `(a % b + b) % b` to always get a positive result.

There also exist the increment (++) and decrement (--) operators, as adding or subtracting one is a common operation. There is a post increment `expr++` and a pre increment `++expr`. The difference is that the post increment increments the value before it is used in the expression, while the pre increment returns the new value after the expression is evaluated.

Lastly, arthimetic assignment operators like `+=, -=, *=, /=, %=` which are semantically equivalent to `a = a + b` but more concise exist.

4 Fundamental Types

4.1 Integers

Integers have a maximal and minimal value, which can be found by using the `std::numeric_limits<T>::min()` and `max()` function respectively. The type `int` is represented by B bits and comprises the set of values

$$\{-2^{B-1}, -2^{B-1}+1, \ldots, -1, 0, 1, \ldots, 2^{B-1}-1\}$$

For most platforms $B = 32$ however C++ only guarantees $B \geq 16$. Arithmetic operations leading to values outside this range lead to undefined behaviour.

There also exist unsigned integers, which only represent positive values from 0 to $2^B - 1$ and can be declared with the `unsigned` keyword and suffixing the number with a `u`.

```
unsigned int a = 42u;
```

However in this course, we go by the practice to not use unsigned integers, unless absolutely necessary.

When using operands of diffrent types, C++ converts to the more general type, in case of a signed and unsigned integer, to the unsigned type. The conversion in this case is that for $x < 0$, x is converted to $2^B + x$.

4.2 Floating Point Numbers

So far we have seen the integer division which occurs when dividing two `int`s. To enable fractional numbers, we have to use the type `float`.

4.3 Fixed and Floating Point Numbers

Floating point numbers are represented in a way similar to scientific notation.

Definition 4.1: Floating point representation

A floating point number is represented as

$$x = m \cdot b^e$$

where m the MANTISSA and e the EXPONENT.

In scientific notation, the base is usually 10.

4.4 Floating Point Numbers in C++

In C++, there exist the types `float` and `double` approximations to real numbers. `float` has approximately 7 decimal digits for the mantissa and an exponent up to ± 38. `double` has 15 digits for the mantissa and an exponent up to ± 308.

Floating point literals are automatically of type `double`, unless we add the suffix `f`.

The operators for floating point numbers are the same as for integers, however the division operator (/) now performs a "proper" division and no modulo operator (%).

As floating point numbers are more general than integers, when using operands of both types, the integer is converted to a floating point number.

As a warning, floating point numbers do not represent $\mathbb{R}$ completely, but only a finite subset of it. This can lead to unwanted behaviour.

4.5 Logical Values

Logical values are represented as values 0 (false) and 1 (true). The type representing logical values is `bool`. There are also the literals `true` and `false`.

C++ further offers the binary relational operators

- `==` (equal),
- `!=` (not equal),
- `<` (less than),
- `<=` (less than or equal),
- `>` (greater than), and
- `>=` (greater than or equal),

which all return a `bool` value.

Booleans can be combined and the canonical operators are

- the conjunction `&&` (logical AND),
- the disjunction `||` (logical OR), and
- the negation `!` (logical NOT).

The precedence of these operators is that `!` has the highest precedence, followed by `&&` and lastly `||`.

Combining this with other operators, we get the following order:

1. Unary logical NOT `!`
2. Binary arithmetic operators (`*`, `/`, `%`)
3. Relational operators (`<`, `<=`, `>`, `>=`)
4. Binary logical Operators (`&&`, `||`)

An important note is that with theese relational operators, we have completeness.

Theorem 4.2:

Any binary logical operator can be expressed by the three canonical operators `!`, `&&`, and `||`.

The types `bool` and `int` are interchangeable, where `false` is converted to 0 and `true` to 1. Similarly, 0 is converted to `false` and any non-zero value to `true`. However, using integers instead of booleans is discouraged.

On a last note, C++ performs SHORT-CIRCUIT EVALUATION. This means that in an expression like `a && b`, if `a` is false, `b` is not evaluated.

5 Control Statements

5.1 Selection Statements

We begin by introducing statements that allow to choose between diffrent branches of code to execute.

Definition 5.1: if-statement

The IF-STATEMENT allows us to execute a block of code only if a certain condition is met.

```
if (condition) { /* code */ }
```

where `condition` is a boolean expression.

Similarly, we can add an `else` branch to execute code if the condition is not met.

Definition 5.2: if-else statement

The IF-ELSE STATEMENT allows us to execute one of two blocks of code depending on whether a condition is met or not.

```
if (condition) {
    /* code if condition is true */
} else {
    /* code if condition is false */
}
```

It is good practice to indent the code inside the blocks to improve readability.

5.2 Iteration Statements

Iteration statements implement loops. The most common loop is the `for`-loop.

Definition 5.3: for-loop

The FOR-LOOP allows us to execute a block of code multiple times, with a loop variable that changes its value in each iteration.

```
for (initialization; condition; expression) {
    /* code */
}
```

The `for`-statement has the following semantics:

1. The `initialization` is executed once.
2. The condition is evaluated. If it is false, the loop ends.
3. The code block is executed.
4. The `expression` is executed.
5. Go to step 2.

Loops run the danger of becoming infinite loops. To prevent this, it is important that the expression is next to the condition instead of at the end of the loop.[1]

[1] In general, it is not possible to decide whether a program will terminate or not. This is known as the Halting Problem.

5.3 Blocks

Blocks are used to group multiple statements together. They are defined by curly braces `{ }`. Most often, blocks are used in control statements to define the code that should be executed conditionally or repeatedly, but in principle they can be used anywhere.

To avoid errors, it is good practice to always use blocks in every selection or iteration statement.

5.4 Visibility

Variables defined inside a block are only VISIBLE inside that block and its sub-blocks. This is known as SCOPE. Trying to access a variable outside its scope will result in a compilation error. The SCOPE OF A VARIABLE starts after its declaration and ends when the current scope ends.

There are exceptions however, where variables in a scope are not visible in sub-scopes. This can happen when a variable in a sub-scope is defined with the same name as a variable outside the scope.

Variables that are defined inside a block are called LOCAL VARIABLES. Variables that are defined outside of any block are called GLOBAL VARIABLES and are visible everywhere.

Local variables are recreated each time their declaration is reached (ALLOCATION) and destroyed when their scope ends (DEALLOCATION). This can lead to unwanted behavior inside loops.

5.5 While Loop

Besides the `for`-loop, there is also the `while`-loop.

Definition 5.4: while-loop

The WHILE-LOOP allows us to execute a block of code repeatedly as long as a condition is met.

```
while (condition) {
    /* code */
}
```

The semantics are as follows:

1. The condition is evaluated. If it is false, the loop ends.
2. The code block is executed.
3. Go to step 1.

5.6 Do-While Loop

If we instead want to execute the code block at least once, we can use a `do-while`-loop.

```
do {
    /* code */
} while (condition);
```

Notice the semicolon at the end of the `do-while`-loop. It is important.

5.7 Jump Statements

To leave a loop early, we can use a `break`-statement.

Definition 5.5: break statement

The BREAK STATEMENT immediately exits the innermost loop or switch statement.

Similarly, we can use a `continue`-statement to skip to the end of the loop, however not exiting it and continuing with the next iteration.

Both theese jump instructions can help to avoid deeply nested if-else constructs, at the cost of complicating the control flow of the program.

5.8 Control Flow and Flow Diagrams

To discuss how a program executes, the notion of control flow is useful.

Definition 5.6: Control Flow

The CONTROL FLOW of a program is the order of the execution of its statements.

Control flow can be visualized using flow diagrams. A flow diagram for an if-else statement is shown in figure 3.

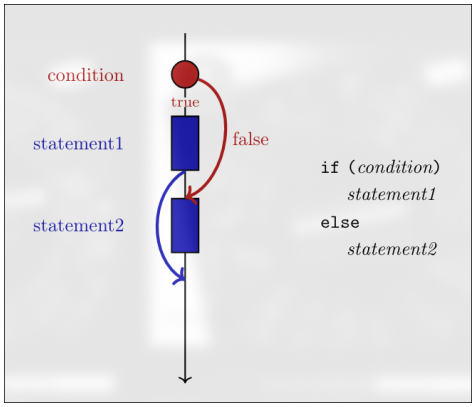

Figure 3: Control Flow Diagram of an if-else statement

When looking at flow diagrams, we can see that we essentially only need an if statement and a goto statement to express any control flow.

When choosing which control statements to use, there should be a balance between complexity and readability.

5.9 Switch Statement

Instead of using multiple if-else statements, we can also use a `switch`-statement.

Definition 5.7: Switch Statement

The SWITCH STATEMENT allows us to choose between multiple branches of code to execute based on the value of an expression.

```
switch (expression) {
    case constant1:
        /* code */
        break;
    case constant2:
        /* code */
        break;
    ...
    default:
        /* code */
}
```

The `expression` is evaluated once and compared against the constants in each `case`. If a match is found, the corresponding block of code is executed. The `break` statement is used to exit the switch statement. If no match is found, the `default` block is executed if it is present.

In C++, it is possible to have multiple cases fall through to the same block of code by omitting the `break` statement.

```
switch (value) {
    case 1:
    case 2:
        // code for both case 1 and 2
        break;
    case 3:
        // code for case 3
        break;
}
```

6 Functions

Functions are used to group code that performs a frequently used task.

6.1 Defining Functions

A function is defined by specifying its return type, name, and parameters.

Definition 6.1: Function Definition

A function definition has the following syntax:

```
T fname (T_1 pname_1, T_2 pname_2, ..., T_n pname_n) {
    /* code */
}
```

where:

- `T` is the return type of the function.
- `fname` is the name of the function.
- `T_i` is the type of the i-th parameter.
- `pname_i` is the name of the i-th parameter.

Parameter names are the names used to refer to the parameters inside the function body.

Functions can be defined without a seperator (semicolon) in between them, however, they cannot be defined locally.

As an example, consider the following function.

```
float Harmonic(int n){
    float res = 0;
    for (int i = 1; i <= n; i++){
        res += 1.0/i;
    }
}
```

6.2 Function Calls

Functions are called by

```
fname(expression_1, expression_2, ..., expression_n);
```

where `expression_i` is an expression that evaluates to a value. These values must be convertible to the types of the corresponding parameters.

Currently it holds that all arguments are R-values as well as the function call itself. This behaviour is called PASS-BY-VALUE.

6.3 Evaluation of Function Calls

When a function is called, the following steps are performed:

1. The arguments of the call are evaluated,
2. The formal parameters are initialized with the values of theese evaluations,
3. The body is executed, the formal parameters are treated as local variables,
4. The function returns (`return result;`)
5. The return value yields the value of the function call.

Notice that the formal parameters are local variables, so one can have a variable `x` in the main program and a parameter `x` in the function, they are different variables and changing `x` in the function does not alter the one in the main program. Furthermore, each function call declares a new set of local variables.

6.5 The type void

Functions that do not return a value and only have some side effects have the return type `void`. No `return` statement is needed in this case. However, one can still use `return;` to exit the function early.

Non-void functions have undefined behaviour if they do not return a value.

6.6 Pre- and Postconditions

To make the intended use of functions clear, we write pre- and postconditions. A PRE-CONDITION specifies what must be true when calling the function and defines the DOMAIN of the function.

A POST-CONDITION meanwhile specifies what is guaranteed to be true after the function has been executed, assuming the pre-condition was true.

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e){
    /* code */
}
```

In general, pre-conditions should be as weak as possible, while post-conditions should be as strong as possible.

The standard library `#include <cassert>` can check pre-conditions at runtime using the `assert` macro. If the expression passed to `assert` evaluates to false, the program is aborted. This can be very powerful for debugging.

```
#include <cassert>
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e){
    assert(e >= 0 || b != 0.0); // PRE
    /* code */
}
```

6.7 Stepwise Refinement

To solve complex problems, one can use STEPWISE REFINEMENT.

Definition 6.2: Stepwise Refinement

Stepwise refinement is the process of breaking down a complex problem into smaller subproblems, solving each subproblem individually, and combining the solutions to solve the original problem.

This is done by writing a very abstract solution and having functions only consisting of comments. Then, each comment is replaced by a function that implements the described functionality.

Consider the following example:

Example 6.3: Two Rectangles

Given two rectangles, determine whether they overlap.

Solution. We start with the most abstract solution:

```
int main(){

    // Input 2 rectangles

    // Is there an overlap?

    // Output result

    return 0;
}
```

Now we replace each comment with a function that implements the described functionality. Then we could repeat the process for each function.

6.8 Scope

The SCOPE of a function starts at its declaration and ends at the end of the file. However, functions can be DECLARED before they are DEFINED. The process is also called FORWARD DECLARATION.

Definition 6.4: Function Declaration

A function declaration has the following syntax:

```
T fname (T_1 pname_1, T_2 pname_2, ..., T_n pname_n);
```

Notice the semicolon at the end.

This is useful when we have two functions that call each other respectively.

6.9 Libraries

As seen before, functions are extremely handy. Some functions are so useful in fact, that they are used in many different programs. To avoid having to change the same function in many different programs, we can put them in a LIBRARY.

This is done by putting the function definitions in a separate file, let's say `mylib.cpp`. Then, in the main program, we can include the file using the `#include "mylib.cpp"` instruction.

This still has some downsides, as the whole file is copied into the main program. Thus the function is compiled every time we include the file and we cannot import the library in multiple files without causing redefinition errors.

To solve this, we can use HEADER FILES. A header file contains only the function declarations of the functions in the library. It is then included in the main program. Header files usually have the extension `.h`.

6.10 Name Spaces

In large projects, it can happen that two libraries define functions with the same name. To avoid this, C++ has NAME SPACES.

Example 6.5: Standard Library

The standard library is in the `std` namespace. Thus, to use the `cout` object, one has to write `std::cout`.

A namespace is defined as follows:

```
namespace nsname {
    /* function declarations */
}
```

In principle, one can also use the `using namespace nsname;` instruction to avoid having to write `nsname::` before each function call. However, this is not recommended!

6.11 The C++ Standard Library

The C++ standard library is used to avoid reinventing the wheel. It contains many useful functions and data structures. Details can be found on https://cppreference.com.

7 Vectors

Vectors are dynamic arrays that can change their size during runtime. Formally, a vector is given by the expression `std::vector<T> v;` where `T` is the type of the elements in the vector.

A vector can be initialized with a size and an initial value for all elements:

```
std::vector<int> v(10, 0); // vector of size 10, all elements initialized to 0
```

Elements in a vector can be accessed using the SUBSCRIPT OPERATOR `[]`, both for reading and writing.

Indexing starts at 0, so the first element is at index 0, the second at index 1, and so on.

7.1 Memory Layout and Properties

Vectors support random access, meaning that accessing an element at a specific index is very efficient. This is because vectors store their elements in a contiguous block of memory, allowing for direct access to any element using its index.

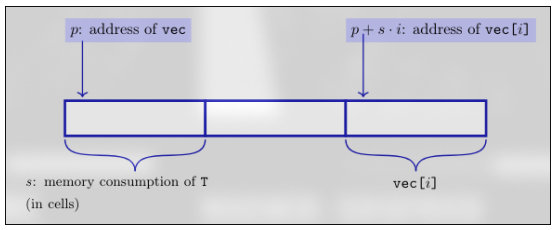

Figure 4: Memory layout of a vector

7.2 Functionality

There are several ways to initialize a vector:

```
std::vector<int> v1; // empty vector
std::vector<int> v2(10); // vector of size 10, default initialized (0 for int)
std::vector<int> v3(10, 5); // vector of size 10, all elements initialized to 5
std::vector<int> v4 = {1, 2, 3, 4, 5}; // vector initialized with a list of values
```

When accessing elements, be aware that accessing an out-of-bounds index leads to undefined behaviour. *It is the sole responsibility of the programmer to ensure that indices are valid.* This can be prevented by using the `at()` method, which throws an exception if the index is out of bounds. However, one still should avoid out-of-bounds access.

Vectors also feature the the following functionalities:

- `v.size()` returns the number of elements in the vector,
- `v.push_back(value)` adds an element to the end of the vector
- Additional functionalities will be covered later.

When using the `v.size()` method, be aware that the return type is an unsigned integer type. This means that if you compare it to a signed integer type, you might get unexpected results.

Example 7.1: **Infinite Loop?!**

Consider the following code snippet:

```
std::vector<int> v = {1, 2, 3};
for (int x = 0; v.size()-x-1 >= 0; x++){
    std::cout << x;
}
```

This is an infinite loop! The reason is that `v.size()` returns an unsigned integer type. And thus the expression `v.size()-x-1` is also of an unsigned integer type. and thus it is always greater than or equal to 0.

This is most easily fixed by changing the loop condition to `x < v.size();` or forcing signed integer arithmetic by using `int(v.size())`.

7.3 Nested Vectors

Storing multi-dimensional data like matrices can be done using nested vectors. A 2D vector can be defined as follows:

```
std::vector<std::vector<T>> matrix;
```

where `T` is the type of the elements in the matrix. Elements can be accessed using two indices: like `matrix[i][j]` to access the element in the i-th row and j-th column.

The initialization of a 2D vector is mostly done using one of the following two options:

```
std::vector<std::vector<int>> matrix1(3, std::vector<int>(4, 0));
// 3x4 matrix initialized to 0
std::vector<std::vector<int>> matrix2 = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
}; // 3x3 matrix initialized with values
```

Functions from vectors translate accordingly. For example, to get the number of rows in a 2D vector, you can use `matrix.size()`, and to find the number of columns in a specific row, you can use `matrix[i].size()`.

Typically, working with 2D vectors involves 2 nested loops to iterate through all elements.

In general, when passing vectors to functions, prefer const references to avoid unnecessary copying.

In general, two dimensional vectors do not to represent rectangular matrices. Each row can have a different number of columns.

To avoid having to permanently write nested versions of `std::vector<T>`, one can use a TYPE ALIAS:

```
using name = type;
```

For example we can define a type alias for a 2D vector of integers as follows:

```cpp
using Matrix = std::vector<std::vector<int>>;
Matrix matrix(3, std::vector<int>(4, 0)); // 3x4 matrix initialized to 0
```

There also exists the type `auto`, which can be used to let the compiler deduce the type of a variable from its initializer. However, be cautious when using `auto` and only use it when the type is obvious from the context and makes the code significantly shorter.

8 Reference Types

Per default, C++ uses VALUE TYPES, meaning that variables hold their own copy of the data. We now look at another type of variable, the REFERENCE TYPE.

Definition 8.1: Reference Type

Reference types are defined as `T&`, where `T` is any valid type.

We can look at a reference type as an alias for another variable.

A reference type differs from a variable in its initialization and assignment.

When initializing a reference type, it must be initialized with a L-value. When assigning a value to a reference type, the value is is also assigned to the original variable.

```cpp
int& darth_vader = anakin_skywalker; // ok
int& j; // error: reference must be initialized
int& k = 5; // error: 5 has no address in memory
```

8.1 Pass by Reference/Value

Per default, C++ uses PASS-BY-VALUE semantics. This means that when passing an argument to a function, a copy of the argument is made.

If instead we want to pass a reference to the original argument, we can use PASS-BY-REFERENCE semantics. This is done by using the reference operator `&` in the function parameter list.

```cpp
void increment(int &x) {
    x = x + 1; // modifies the original argument
}
void decrement(int x) {
    x = x - 1; // modifies only the copy
}
int main() {
    int a = 5;
    increment(a); // a is now 6
    decrement(a); // a is still 6
}
```

Pass by value initializes the passed R-value and creates a copy of it. Pass by reference creates a reference to the original L-value and becomes an alias for it.

8.2 Return by Reference and Temporary Objects

Functions can also return references. This can be useful when we want to pass a function as an argument to another function. This is done by using the reference operator `&` in the return type of the function.

```cpp
int& func();
```

This is called RETURN BY REFERENCE. However, one must be cautious when returning references. If a function returns a reference to a local variable, the reference becomes dangling as soon as the function exits. This leads to undefined behaviour when the reference is used.

Thus, such a function should only return references to variables that outlive the function call, such as static variables or variables passed by reference to the function.

In short: pass by reference can be used to enable effects on arguments and return by reference can be used to enable functions as l-values.

8.3 Const References

Function parameters can be declared as CONST REFERENCES to prevent modification of the original argument. This is especially important if we pass a constant object to a function, as the compiler will not know if the function modifies the object or not and thus throw an error.

```
void print(const std::vector<int>& v) {
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    // v[0] = 5; // error: cannot modify a const
    ↪ reference
}
```

In this case, even a R-value can be passed to the function. The compiler will generate a temporary object that is passed to the function.

In general, declare function arguments that are not fundamental types as `const T&`.

9 Characters and Texts

A character can be represented by the fundamental type `char`. A text is a value of the type `std::string`, which behaves like a vector of characters.

The type `char` is formally an integer type, meaning that it is convertible to `int` (and `unsigned int`). Furthermore, arithmetic operations can be performed on characters. In general, characters occupy 8 bits of memory.

When using the type `std::string`, the header `<string>` must be included. Strings can be initialized in multiple ways:

```
std::string s1 = "Hello, World!";
std::string s2 = std::string(n, 'a'); // n times 'a'
```

Standard operations on vectors also work on strings, such as

- `s.size()` returns the number of characters in s,
- `s[i]` accesses the i-th character (0-indexed),
- `s + t` concatenates two strings `s` and `t`.

When working with character manipulation, sometimes it is useful to allow whitespace characters (spaces, tabs, newlines) to be part of the input. This can be done using `std::cin >> std::noskipws;`.

9.1 ASCII and UTF8

The ASCII code defines a mapping from characters to integers and is supported on all common systems. For letters and digits, the mapping is as follows:

Character	Integer
'A', 'B', ..., 'Z'	65, 66, ..., 90
'a', 'b', ..., 'z'	97, 98, ..., 122
'0', '1', ..., '9'	48, 49, ..., 57

The printable ASCII characters are those with integer values between 32 and 126 (inclusive).

The conversion to integers can be used to iterate over characters

```
for (char c = 'a'; c <= 'z'; ++c) {
    std::cout << c << ' ';
    // prints: a b c d e ... z
}
```

9.2 Example: Caesar Cipher

A simple encryption method is the Caesar cipher.

Example 9.1: **Caesar Cipher**

Implement a program that reads a text and an integer shift value, and outputs the text with each letter shifted by the given value.

```
char shift(char c, int s){
    if ( c >= 32 && c <= 126 ) {
        c = (c - 32 + s) % (126 - 32 + 1) + 32;
    }
    return c;
}
```

10 Recursion

Many mathematical functions are naturally defined in a recursive way. For example, the factorial function $n!$ is defined as

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot (n-1)! & \text{if } n > 0. \end{cases}$$

In C++, this can be implemented as follows:

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

If the user calls `factorial(3)`, the following sequence of calls occurs: factorial(3) → factorial(2) → factorial(1).

In general, a recursive function is defined by as follows

Definition 10.1: Recursive Function

A recursive function is a function that calls itself

```
void f(){
    // ...
    f(); // function calls itself
    // ...
}
```

The above function itself would lead to an INFINITE RECURSION, which is even worse than an infinite loop as it burns time and memory.

To write useful code, we thus *need* to guarantee termination. This is done by defining a BASE CASE that does not call the function itself. Any reasonable recursive function will thus always contain an if statement.

10.1 Call Stack

Local variables and additional information about function calls are stored on the CALL STACK. Consider the following simple non-recursive function:

```
int f(int x){
    int y = 2 * x;
    return y;
}
```

When this function is called, the function argument as well as the return adress (where to continue after the function call) are pushed onto the stack. The function then uses the value on the call stack to initialize its formal argument `x` and executes its body. The local variable `y` also lives on the call stack.

When the function returns, the program jumps back to the return address and the function call is replaced by the return value. All elements above the return address are popped from the stack.

This works similarly for recursive functions, except that now we push function calls onto the stack multiple times. For example, when calling `factorial(3)`, this pushes three function calls onto the stack.

After reaching the return statement of the base case, the function calls are popped from the stack one by one, returning their values to the previous function call. This is illustrated in the following figure.

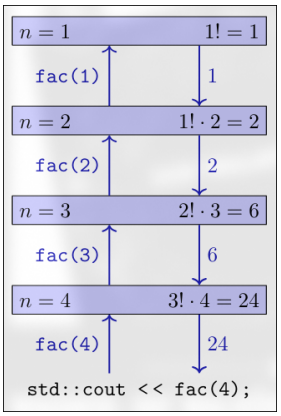

Figure 5: Call stack during execution of factorial(4)

Note that each function call has its own copy of the local variables, including formal arguments.

10.3 Example: Bitstrings

So far we've seen recursion in the context of mathematical functions, where it made the task somewhat harder. But recursion is often a key to simplify problems that are hard to solve otherwise.

Example 10.2:

Generate all bit strings of length n. For example, for n=3, the output should be

```
000
001
...
110
111
```

Solution. We start by a systematic approach. We could start by an empty string and then in each step, we either append a '0' or a '1' to the current string until we reached n bits.

Doing this iteratively is possible, but quite cumbersome, as we would need to keep track of all the strings we are currently building.

The key question to ask is: *What subproblems do exist such that if we had the solutions for these subproblems, the overall solution would become easy?"*

For this problem, this might be that if we know how to generate all bit strings of length n starting with '0' and all bit strings of length n starting with '1', then we can just combine these two lists to get all bit strings of length n. But how do we generate these two lists? We can use the same idea again: If we know how to generate all bit strings of length $n-1$, then we can just prepend '0' to each of these strings and so on.

The next question is: *What subproblem is trivial/easy or, in other words, when does the recursion stop?"*

In this case, the answer is that there is only one bit string of length 0, the empty string.

All together, we get the following recursive function:

```cpp
void all_bit_strings(int n, std::string prefix){
    if (int(prefix.size()) == n) {
        std::cout << prefix << '\n';
    } else {
        all_bit_strings(n, prefix + '0');
        all_bit_strings(n, prefix + '1');
    }
}
```

Our code works, however there is one issue. We are using a lot of memory and it may happen that we run out of stack space for large n. This is called a STACK OVERFLOW.

In our case, this can be fixed by avoiding to pass the prefix by value, and instead passing it by reference. This way, we do not create a new copy of the prefix for each function call, but instead modify the same string. We then need to keep track of the length of the prefix separately.

```cpp
void all_bit_string(std::string& prefix, int i)
```

10.5 General Recursive Problem Solving Strategies

There exist two main strategies to solve problems recursively. For this we consider the problem to calculate the sum of all elements in a vector of integers.

DECREASE AND CONQUER: For this strategy, we reduce the problem by splitting it into the subproblem of calculating the sum of the last $n-1$ elements and then adding the first element to this sum. The base case is when the vector is empty, in which case the sum is 0.

DIVIDE AND CONQUER: For this strategy, we split the vector into two halves, calculate the sum of each half recursively and then add the two sums together. The base case is when the vector has one or zero elements, in which case the sum is the element itself or 0, respectively. Often when using this strategy, a index `from` and `to` is passed to indicate the part of the vector to consider.

In summary, decrease and conquer removes one element from the problem while divide and conquer splits the problem into two smaller subproblems.

In general, divide and conquer is more efficient, as it tends to only have recursion depth $\log_2(n)$, while decrease and conquer has recursion depth n and thus has higher risk of stack overflow.

10.6 Recursion and Iteration

Algorithms that can be expressed recusively may look like the following

```cpp
int fib(int n){
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

This code has one major issue: It is very inefficient. The reason is that the same function calls are performed multiple times. For example, `fib(5)` calls `fib(4)` and `fib(3)`, but `fib(4)` also calls `fib(3)` and `fib(2)`. So in total, the base case is called 5 times! Or in general, `fib(1)` is called fib(n) times, which is exponential in n.

We will not discuss how to fix this here, but it is important to note that any recursive function can be rewritten in an iterative way using loops. This is often more efficient, but also more complicated.

11 Number Representations

We now want to look at how numbers are represented in a computer.

11.1 Integer Numbers

Computers use a BINARY NUMBER REPRESENTATION to represent numbers.

Definition 11.1: Binary Number Representation

A binary representation of an integer consists of two possible values for each digit, namely bits from the set $\{0, 1\}$.

The representation $b_n b_{n-1} ... b_1 b_0$ represents the number

$$b_n \cdot 2^n + b_{n-1} \cdot 2^{n-1} + ... + b_1 \cdot 2^1 + b_0 \cdot 2^0 = \sum_{i=0}^{n} b_i \cdot 2^i.$$

In C++, binary numbers can be written with the prefix `0b`, thus `0b101011` represents the number 43.

As binary numbers are pretty hard to read, we often use the hexadecimal representation, which uses the digits from 0 to 9 and the lettes a to f. They are written with the prefix `0x`, thus `0xff` represents the number 255.

The following table can be used to convert between binary the number systems Hexadecimal numbers are used in for

Table 1: The digits of the number systems

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	a
3	0011	3	11	1011	b
4	0100	4	12	1100	c
5	0101	5	13	1101	d
6	0110	6	14	1110	e
7	0111	7	15	1111	f

a variety of applications, for example to represent colors where colors are encoded as #RRGGBB.

11.2 Floating Point Number Systems

Before we look at the specifics of floating point numbers, we first define the following

Definition 11.2: Floating Point Number System

A floating point number system $F(\beta, p, e_{\min}, e_{\max})$ is defined by a base β, a precision p and an exponent range $[e_{min}, e_{max}]$.

A floating point number in this system is represented as

$$\pm(d_0.d_1 d_2 ... d_{p-1})_b \cdot b^e$$

where $d_0, d_1, ..., d_{p-1}$ are digits in base b and $e_{min} \leq e \leq e_{max}$.

At this point it is important to note, that the representation of a number is not unique. For example, the number 1.0 can be represented in base 10 as

$$1.0 \cdot 10^0 = 0.1 \cdot 10^1 = 0.01 \cdot 10^2 = ...$$

This is fixed by using a NORMALIZED REPRESENTATION, which requires that $d_0 \neq 0$ if the number is non-zero. It is denoted by $F^*(\beta, p, e_{min}, e_{max})$.

In this system, there are some numbers that cannot be represented. For example, 0 cannot be represented in a normalized representation. Similarly, the smallest absolute value that can be represented is

$$\min = 1 \cdot \beta^{e_{min}}$$

Calculations are performed as one might expect; first the numbers are denormalized, after which the operation is performed and finally the result is normalized again and rounded to the correct precision.

11.2.1 IEEE 754 Standard

The IEEE Standard 754 defines floating point numbers and their rounding behavior for almost every common programming language. They define SINGLE PRECISION (float) and DOUBLE PRECISION

The type float corresponds to $F^*(2, 24, -126, 127)$ and requires 32 bits of storage. Made up of 1 sign bit, 23 mantissa bits (where the leading does not need to be stored) and 8 exponent bits with 254 possible exponents and two for special values.

The type double corresponds to $F^*(2, 53, -1022, 1023)$, which requires 64 bits of storage. Made up of 1 sign bit, 52 mantissa bits and 11 exponent bits with 2046 possible exponents and two for special values.

All arithmetic operations first compute the *exact* result and then are rounded to the nearest representable number.

11.2.2 Decimal and Binary Systems

Humans like base 10, while computers like base 2. Thus the computer has to be able to convert between the two systems. Let us look at the conversion of a decimal number $0 < x < 2$.

We can check if $x \geq 1$. If this is the case, the first digit is 1, else 0. We then subtract the value of the first digit and multiply the result by 2. We repeat this process until we have computed p digits.

Doing this for values like 0.1_{10} results in $0.0\overline{0011}_2$, which is a repeating binary fraction. This number cannot be represented and thus gets approximated. This can lead to CONVERSION ERRORS, which explains, why `1.1-0.1!=1.0` in C++.

11.3 Floating Point Rules

When working with floating point numbers, there are three rules that should be followed:

1. Do not test floating point numbers for equality.

2. Do not add floating point numbers with very different magnitudes.
3. Do not subtract nearly equal floating point numbers.

12 Custom Data Types

The goal of this section is to build a C++ type `rational` that represents fractions of integers. We want to be able to add two rationals, as well as input and output them.

12.1 Struct Declaration

A STRUCT is a custom data type that can hold multiple values. It is declared as follows

Definition 12.1:

A struct is declared with the keyword `struct` followed by the name of the new type and a block containing the MEMBERS of the struct.

```
struct T {
    T_1 name_1;
    T_2 name_2;
    ...
    T_n name_n;
};
```

where T is the name of the new type, T_i are types of the members and $name_i$ are the names of the members.

The range of a struct is given by the cartesian product of the ranges of its members.

A member of a struct t with the name $name$ is accessed with `t.name`.

For our example, we define a struct `rational` as follows

```
struct rational {
    int n;
    int d;
};
```

We will enforce $d \neq 0$ later.

Member variables can be of the type of another struct, and don't nessecairly all have to be of the same type.

12.2 Struct Initialization

A struct can be initialized just as any built in type. However, the variable gets initialized even when just declaring it, opposed to fundamental types.

If we want to directly initialize the members of a struct, we can use AGGREGATE INITIALIZATION. This is done by providing a list of values in curly braces.

```
rational s = {5,2};
```

Another way is to make a memberwise copy of another struct

```
rational t = s;
```

12.3 Assignment

After initialization, structs can be assigned as values as normal. We can also directly assign to members of a struct.

```cpp
rational r; // Default initialization
r = {3,4}; // Aggregate initialization
r.n = 5; // Assign to member
std::cout << r.n << " / " << r.d << "\n"; // Access members
```

12.4 Structs as Function Arguments

Just like fundamental types, structs can be used as function arguments or return types. They can as usual be passed by value or by reference if needed. The same principles as with fundamental types and vectors apply.

12.5 Function and Operator Overloading

Instead of having a function `add(rational, rational)`, we would like to use the `+` operator to add two rationals. This is done by OPERATOR OVERLOADING.

Before we dive into operator overloading, we first look at function overloading. In C++, a function is defined by its SIGNATURE, which consists of its name and the types of its parameters. *The return type is not part of the signature.* Two functions can have the same name as long as their signatures are different. This is called function overloading.

```cpp
double sq(double x) { return x*x; }
int sq(int x) { return x*x; }
```

The compiler will then choose the "correct" function to use.

Operators are nothing else than special functions. For example, `+` is a function with the name `operator+`. For most primitive types, these operators are already defined. However, we can also define them for our own types. This is called operator overloading.

```cpp
rational operator+(const rational& a, const rational& b) {
    return {a.n * b.d + b.n * a.d, a.d * b.d};
}

rational operator-(rational a){
    a.n = -a.n;
    return a;
}
```

This defines the addition of two rationals, which we can now do with `r = a + b;`.

We have also defined the unary minus operator, which which only takes one argument.

We can even go a step further and define arithmetic assignment operators

```cpp
rational& operator+=(rational& a, const rational b) {
    a = a + b;
    return a;
}
```

Here it is essential that we pass `a` *by reference, as we want to modify it.* We also return a reference to the modified object, as this is the standard behavior of these operators, and allows us to write horrible code like `a += b += c;`.

Comparison operators should also be defined as otherwise the compiler yields "error: no match for 'operator==' ". We can define it in our case as

```cpp
bool operator==(const rational& a, const rational& b) {
    return a.n * b.d == b.n * a.d;
}
```

For output, we remember that `std::cout` is also an operator. On its left hand side is an `std::ostream` object, and on the right hand side is what we want to output. Thus we can overload the `operator«` as follows

```cpp
std::ostream& operator<<(std::ostream& out, const rational& r) {
    os << r.n << "/" << r.d;
    return os;
}
```

Again it is crucial to have the first argument as a reference, as well as returning a reference to the output stream.

Similarly, we can overload the input operator `operator»`

```cpp
std::istream& operator>>(std::istream& in, rational& r) {
    char c; // to read the '/'
    return in >> r.n >> c >> r.d;
}
```

There could also be error checking here to ensure that the input is valid. However, this is not discussed here.

We have implemented all the functionality we wanted. One issue remains: We cannot check if a rational is valid, i.e. if its denominator is not zero. We simply cannot avoid this with the tools we have seen so far.

13 Information Hiding

A new type offering functionalities such as `struct rational` should be stored as a library. Again we can split the files into a header file `rational.h` and an implementation file like `rational.cpp`. This separation makes the header file more readable and allows a user to not needing to know how the functions are implemented.

13.1 Encapsulation

Usually, we want to guarantee certain INVARIANTS like $d \neq 0$ for a rational number. Similarly, we may want to make sure that a rational number is always in reduced form. (REPRESENTATION)

In general, users should be provided with the "what" and not the "how". This is called INFORMATION HIDING or ENCAPSULATION.

> Definition 13.1: **Information Hiding Principle**
>
> A type is uniquely defined by its possible values and its functionality. The representation should not be accessible.

In C++, this is offerd by CLASSES.

13.2 Classes

Classes are a variant of structs and provide the concept for encapsulation.

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

In C++, it can be controlled if members are accessible outside a struct or class by the modifiers `public` and `private`.

The only major difference between a struct and a class is that by default, structs hide nothing, while classes hide everything.

```
struct S {
    private:
        int x; // private member
        int y; // private member
};
```

The above would be equivalent to a class with the variables x and y. To make use of this we need another concept.

13.3 Member Functions

MEMBER FUNCTIONS are like members of a class and have access to hidden data. Theese functions can also be made public.

```
class rational {
    public:
        int numerator() const {
            return n;
        }
        int set_denominator(int value){
            assert(value != 0);
            d = value;
        }
    private:
        int n;
        int d; // INV: d != 0
};
```

Users now cannot violate the invariant $d \neq 0$ unless for initialization, which will adress below.

Member functions can be accessed just like every other member.

One thing to note is the `const` after the function name. This means that the function does not modify the object. This is important as objects which are `const` can only call `const` member functions.

When a member function is called, the object itself is implicitly passed as a parameter. This parameter is called `this` and is a pointer to the object itself. Thus, inside a member function, the members of the object can be accessed directly. In fact, in the above example, `return n;` is a shorthand for `return this->n;`.

13.4 Constructors

Constructors are special member functions named like the class itself. For our example, it could look like this:

```
class rational {
    public:
        rational(int num, int den) : n(num), d(den) {
            assert(denominator != 0);
        }
        ...
};
```

Constructors are diffrent from normal member functions in the way that they do not return a type and provide a way to initialize an object outside it, before executing any other code.

Writing the constructor as above, initializes the members `n` and `d` before executing the body of the constructor.

The DEFAULT CONSTRUCTOR is a constructor that takes no parameters. If a declaration without arguments is meant to compile, it must exist.

If there are no constructors defined, the compiler provides a default constructor which initializes all members with their default values. *This can result in no initialization for fundamental types.*

```
class rational {
    public:
        rational() : n(0), d(1) {} // default
        ↪ constructor
        ...
};
```

If specifically wanted, the default constructor could be deleted with

```
rational() = delete;
```

Their can also exist user defined conversions for example we can add a constructor that takes only one integer and sets the denominator to 1.

```
rational(int num) : n(num), d(1) {}
```

If we want to output a rational number as a double, we can add a member function such as

```
operator double() const { return double(n)/d; }
```

13.5 In-Class vs. Out-of-Class

In order to have a header file with only the interface, the member functions can be defined outside the class using the DOMAIN RESOLUTION OPERATOR `::`.

```
class A {
    public:
        void f() const;
        int g(int x); // declaration
        ...
};

//Definition outside the class
void A::f() const {
    ...
}
int A::g(int x) {
    ...
}
```

14 Container and Iterators

A CONTAINER is a data structure that stores multiple values. Together with ITERATORS, they provide a way to access the stored values.

14.1 Containers

Viewed abstractly a vector is a ordered collection of elements plus a set of operations. In C++, the `std::vector<T>` class and similar data structures are called containers.

Each character has diffrent characteristics and nearly all have their use cases. For example, `std::list<T>` is efficient at inserting and deleting elements, but accessing elements by index is inefficient.

Another example is `std::unordered_set<T>`, which provides the notion of a mathematical set. It is efficient at checking for membership, but does not provide any order of the elements.

There also exists `std::set<T>`, which is similar to the unordered set, but provides an order of the elements.

14.2 Iterators

If one wants to for example print all elements of a container, one needs to iterate over all elements. This is done using a ITERATOR. Any contained is by convention guaranteed to support the following operations:

- `begin()`: returns an iterator to the first element
- `end()`: returns an iterator to one past the last element
- `++it`: advances the iterator to the next element
- `*it`: dereferences the iterator to access the element

This allows to write code like

```
std::vector<int> v = {1,2,3};
for (std::vector<int>::iterator it = v.begin(); it !=
↪  v.end(); ++it) {
    std::cout << *it << std::endl;
}
```

Since the types of iterators can be long, often times the `auto` keyword is used.

Iterators are useful as they provide a uniform way to access elements of diffrent containers.

The algorithm library in C++ provides many useful algorithms that work with iterators. For example, to find the maximum element in a container, one can use

```
std::vector<int> v = {1,5,3};
auto it = std::max_element(v.begin(), v.end());
std::cout << *it << std::endl; // prints 5
```

Iterators also allow the use of range-based for loops, that loop over all elements of a container.

```
std::vector<int> v = {1,2,3};
for (int x : v) {
    std::cout << x << std::endl;
}
```

For this type of loop, reference types can also be used to modify the elements in place.

14.3 Const Iterators

Every container also provides CONST ITERATORS that do not allow modification of the elements they point to. This is useful when working with `const` containers. In this case the corresponding interators are made like this

```
const std::vector<int> v = {1,2,3};
for (std::vector<int>::const_iterator it = v.begin(); it
↪  != v.end(); ++it) {
    std::cout << *it << std::endl;
}
```

15 Pointers and Dynamic Memory

15.1 Pointers

References are often not sufficient. For example, they always need to refer to an existing object and cannot be changed after initialization. This is where POINTERS come into play.

Definition 15.1: Pointer Type

A pointer type `T*` is a type that stores the address of an object of type `T` or a null pointer.

An expression of type `T*` is called POINTER TO `T`.

A pointer is declared as usual. It is important that a pointer `T*` actually points to an object of type `T`.

The value of a pointer `T*` is the address of an object of type `T`. To properly initialize a pointer, we need to get the address of an object.

Definition 15.2: Address Operator

The address operator `&` applied to an lvalue of type `T` yields the address of that object as an rvalue.

```
int x = 42; int* p = &x; // p points to x
```

To now access the object a pointer points to, we need to use the DEREFERENCE OPERATOR.

Definition 15.3: Dereference Operator

The dereference operator `*` applied to an expression of type `T*` yields an lvalue of type `T` that refers to the object the pointer points to.

```
int x = 42; int* p = &x; int y = *p; // y == 42
```

A special pointer value is the NULL POINTER which does not point to any object. it is denoted by `nullptr`.

```
int* p = nullptr; // p is a null pointer
```

A null pointer must never be dereferenced. It will lead to undefined behavior.

A pointer can also point to a class. For example our rational class. In this case, the usage of the dereference operator has to be done very carefully.

```
rational r(1,2);
rational* p = &r; // p points to r
int num = (*p).numerator(); // dereference p and call
↪  numerator
```

We have to use the parentheses around `*p` as a pointer does not have member functions. To make this easier, C++ provides the MEMBER ACCESS OPERATOR `->`. Therefore, `p->numerator()` is equivalent to `(*p).numerator()`.

This also explains the `this` pointer inside member functions.

Just like with references, pointers can be used for functions with effects. Furthermore, if functions only need to read data, passing a pointer by const reference is also possible. In this case, there are four cases

```
const int p1; // p1 Is a constant integer
const int* p2; // p2 is a pointer to a constant integer
int* const p3; // p3 is a constant pointer to an integer
const int* const p4; // p4 is a constant pointer to a
↪  constant integer
```

Notice how it does not matter if the `const` is before or after an expression.

Similarly to references, pointers are not absolute. The value at an address can change, even if the pointer itself is constant.

15.2 Dynamic Memory

So far, all objects we created had a fixed lifetime. They were created when the scope was entered and destroyed when the scope was left. However, sometimes we need objects that live longer than the scope they were created in. For this, C++ provides the ability to allocate memory ourselves.

This is done using the NEW operator, which creates memory for the given type, initializes it by the matching constructor and returns the address of the created object.

```
rational* p = new rational(1,2); // create a new
↪  rational number
```

We will come back later to see how to properly manage this memory.

16 Dynamic Data Structures

In the following section, we want to look at a first data structure, the LINKED LIST.

16.1 Linked List

A linked list is a container with the following properties:

- Does not not require contiguous memory
- Efficient delete and insert operations
- Inefficient access by index

Linked lists are what gives rise to the `std::list<T>` container.

A linked list is implemented in terms of elements that point to their successor. The last element points to a null pointer.

The main ingredient is a node for the list, which we call a LNODE. A lnode contains the data and a pointer to the next node.

```
struct lnode {
    int value;
    lnode* next;

    lnode(int v, lnode* n) : value(v), next(n) {}
};
```

As a consequence a linked list, basiclly consists of a single pointer to the first node. For example

```
class our_list {
lnode* head; // pointer to first node
public:
    our_list(int size);
    void print() const;
    int size() const;
    void push_front(int value);
    void push_back(int value);
    ...
};
```

Now functions like `print` can be implemented by iterating over the nodes starting from `head` until a null pointer is reached.

```
void our_list::print() const {
    for (lnode* n = head; n != nullptr; n = n->next) {
        std::cout << n->value << " ";
    }
}
```

The subscript operator can be implemented by iterating over the nodes until the desired index is reached.

```
int& our_list::operator[](int index) {
    lnode* n = head;
    for (int i = 0; i < index; ++i) {
        n = n->next;
    }
    return n->value;
}
```

Notice that this implementation does not check if the index is valid. In particular, it could lead to undefined behaviour.

To build our list, it is particularly easy to add elements at the front.

```
void our_list::push_front(int value) {
    head = new lnode(value, head);
}
```

Here the `new` operator is used to allocate a new node on the heap. The new node's next pointer is set to the current head, and then the head pointer is updated to point to the new node.

Here it is crucial that `lnode` is allocated dynamically. Otherwise, the node would be destroyed at the end of the function, resulting in undefined behavior when accessing the list later.

The constructor can now also be easily implemented by pushing elements to the front.

```
our_list::our_list(int size) {
    head = nullptr;
    for (int i = 0; i < size; ++i) {
        push_front(0); // initialize with zeros
    }
}
```

When adding a `push_back` function, we need to distinguish the cases where the list is empty and where it is not. In the first case, we simply create a new node and set the head pointer to it. In the second case, the following code can be used.

```
lnode* n = head;
for (; n -> next != nullptr ; n = n->next) {
    // advance to last node
}
n->next = new lnode(value, nullptr);
```

For large lists, this will be inefficient. A possible solution to this might be to store a pointer to the last node in the list class. However, we will not pursue this further here.

16.2 List Iterator

We have thus far implemented a container. Every container should have an iterator to access its elements. It should have the following functionalities:

- Acces the current element with `operator*`
- Advance to the next element with `operator++`
- Compare two iterators for equality with `operator==` and `operator!=`

Additionally we also need the member functions `begin()` and `end()`. We do this by defining a INNER CLASS inside the `our_list` class.

```
class our_list {
public:
    ...
    class iterator {
        lnode* current;
    public:
        iterator(lnode* n); // constructor
        iterator& operator++(); // advance to next element
        int& operator*() const; // access current element
        bool operator!=(const iterator& other) const; //compare
    };
    ...
    iterator begin();
    iterator end();
    ...
};
```

We now implement the functions one by one. The constructor simply initializes the current node.

```
our_list::iterator::iterator(lnode* n) : current(n) {}
```

Advancing the iterator is done by setting the current node to the next node.

```
our_list::iterator& our_list::iterator::operator++() {
    current = current->next;
    return *this;
}
```

Similarly, dereferencing the iterator returns the value of the current node.

```
int& our_list::iterator::operator*() const {
    return current->value;
}
```

Finally, comparing two iterators is done by comparing their current nodes.

```
bool our_list::iterator::operator!=(const iterator& other) const {
    return current != other.current;
}
```

But now, with this functionality, we can implement `begin()` and `end()` in the following way.

```
our_list::iterator our_list::begin() {
    return iterator(head);
}
our_list::iterator our_list::end() {
    return iterator(nullptr);
}
```

Having implemented the iterator, we now have access to all the standard algorithms provided by the C++ standard library.

17 Memory Management

We will now look at how to properly manage dynamic memory. To make our life easier, we look at a STACK. This is a container such as the linked list, but which only allows adding and removing elements from the head, also called top.

17.1 Basic Stack Functionality

We use again the notion of a node defined in the previous section. A stack is essentially a pointer to the head node.

```
class stack {
    lnode* topnode; // pointer to top node
public:
    void push(int value){ // add element to top
        topnode = new lnode(value, topnode);
    }
    void pop(); // remove element from top
    bool empty() const { // check if stack is empty
        return topnode == nullptr;
    }
    int top() const { // access top element
        assert(!empty());
        return topnode->value;
    }
    stack(): topnode(nullptr) {} // Default constructor
};
```

Ther now remains the question of how to implement `pop()`.

17.2 Removing Elements from Stack

A first attempt could be the following.

```
void stack::pop() {
    assert(!empty());
    topnode = topnode->next;
}
```

This does work, as we guarantee that the stack is not empty and thus `topnode->next` is a valid pointer. However, this implementation has a major flaw. We essentially allocate memory for each node using `new`, keyword, but never explicity free it. This is a stark contrast to local variables, which are automatically destroyed when they go out of scope.

To free memory allocated with `new`, we need to use the DELETE operator.

As a guideline, *every* `new` *must have a matching* `delete`.

Ignoring this rule leads to MEMORY LEAKS, where memory is allocated but never freed. Over time, this can lead to a HEAP OVERFLOW, where the program runs out of memory.

We should also be careful when using `delete`. Consider the following code.

```
rational* t = new rational;
rational* s = t;
delete s;
// t is now a dangling pointer
```

After the `delete` statement, `t` is a DANGLING POINTER, as it points to memory that has been freed. Using it in a statement like `t->denominator()` leads to undefined behavior.

Another danger lies in double deletion. This would also lead to undefined behavior.

For our stack, we can now implement `pop()` correctly as follows.

```
void stack::pop() {
    assert(!empty());
    lnode* oldtop = topnode;
    topnode = topnode->next;
    delete oldtop; // free memory of old top node
}
```

Notice that we have to store the old top node in a temporary variable as otherwise we would lose the pointer to it and could not delete it.

17.3 Destructors

Consider the following code.

```
{
    stack s;
    s.push(1);
    s.push(2);
    s.push(3);
}
```

When the scope is left, the stack `s` is destroyed. However, we have not freed the memory allocated for the nodes in the stack. This again leads to memory leaks.

> Definition 17.1: **Destructor**
>
> A destructor of a class `T` is the unique member function with declaration `~T();`. It is called automatically when the lifetime of an object of type `T` ends or if `delete` is applied to a pointer to an object of type `T`.

If no destructor is defined, the compiler provides a default destructor that calls the destructors of all members. If we want something non-trivial to happen, we need to define our own destructor.

For our stack, the default deconstructor is not sufficient, as it simply deletes the pointer `topnode`, but not the nodes it points to. We can implement the destructor as follows.

```
stack::~stack(){
    while (!empty()) {
        pop(); // remove all elements
    }
}
```

17.4 Copy Assignment

Currently, our stack does not support an operation for copying data. Consider for example the following snippet of code.

```
stack s1;
s1.push(1); s1.push(2); s1.push(3);

stack s2 = s1; // copy s1 to s2
```

```
s1.pop(); // remove top element from s1
std::cout << s2.top(); // what is the output?
```

The output is undefined behavior. This is because the default copy assignment simply copies the pointer `topnode` from `s1` to `s2`. Thus, both stacks point to the same nodes. When `s1.pop()` is called, the top node is deleted, leaving `s2.topnode` a dangling pointer.

The problem is as follows: Since we do not actually create a copy of the nodes data but only of the pointer, the two stacks use the same data. There exist three possible solutions to this problem:

1. Create ab actual copy of the data (DEEP COPY)
2. Disallow copying alltogether
3. Implement a shared data structure (not discussed here)

17.5 Copy Constructor

Let us first look at the deep copy approach. For this, we need to define a COPY CONSTRUCTOR.

Definition 17.2: Copy Constructor

A copy constructor of a class `T` is a constructor with declaration `T(const T& t);`. It is called when an object of type `T` is initialized from another object of the same type.

```
T x = t;
T x(t);
```

If we do not define a copy constructor, the compiler provides a default copy constructor that performs a member-wise copy. For fundamental types, this means copying the bit pattern.

For our stack, we can implement the copy constructor as follows.

```
stack::stack(const stack& other) : topnode(nullptr) {
    // copy elements from other to this
    if (other.topnode == nullptr) return; // other is empty

    // copy first node
    topnode = new lnode(other.topnode->value, nullptr);
    lnode* last = topnode;
    // copy remaining nodes
    for (lnode* n = other.topnode->next; n != nullptr; n = n->next) {
        lnode* copy = new lnode(n->value, nullptr);
        last->next = copy;
        last = copy;
    }
}
```

With this copying indeed works correctly.

17.6 Assignment

Assignment has the same issues as copying. Here we overload the assignment operator `operator=`.

Definition 17.3: Assignment Operator

The assignment operator of a class `T` is the overloaded function `operator=` as a member function of `T`. It has the signature `T& operator=(const T& other);`. It has the current object `*this` as left-hand side and `other` as right-hand side. By C++ convention, it returns a reference to `*this`.

There are three differences to copying:

- The object `*this` already exists
- We need to free existing resources of `*this`
- There exists the possibility of self-assignment

When implementing the assignment operator, we first will need to check for self-assignment. If `this` and `&other` are the same, we can simply return `*this`.

Next we implement the so called COPY-AND-SWAP idiom. The idea is to first create a copy of `other` using the copy constructor. Then we swap the data members of `*this` and the temporary copy. Finally, when the temporary copy goes out of scope, its destructor is called, freeing the old resources of `*this`.

```
stack& stack::operator=(const stack& other){
    if (this != &other) { // check for self-assignment
        stack copy = other; // Copy constructor
        std::swap(topnode, copy.topnode); // swap data members
    }   // copy goes out of scope and destructed
    return *this; // Convention
}
```

17.7 Rule of Three

Due to close relationship between destructor, copy constructor and assignment operator, there exists the so called RULE OF THREE.

Theorem 17.4: Rule of Three

If a class declares one of the special member functions destructor, copy constructor and assignment operator, all three should be explicitly declared.

A special way to declare these functions is to use the `=delete` specifier, which prevents the function from being used.

```
class non_copyable {
public:
    non_copyable(const non_copyable&) = delete; // no copy constructor
    non_copyable& operator=(const non_copyable&) = delete; // no assignment
};
```

Notice that the destructor cannot be allowed, but one would not want anyway.

17.8 Shared Pointers

SMART POINTERS are pointers that automatically manage memory. They allow us to *outsource* proper memory man-

agement. There exist two main types of smart pointers in C++: `std::unique_ptr<T>` and `std::shared_ptr<T>`. The former implements exclusive ownership of a resource, which we will not discuss further here.

A shared pointer counts the numbers of pointers pointing to an object an deletes that object only when that counter reaches zero.

Shared pointers are used in the same way as regular pointers, up to creation.

```
#include <memory> // for std::shared_ptr
std::shared_ptr<rational> sp1 =
↪  std::make_shared<rational>(1,2);

rational r = *sp1;

std::shared_ptr<rational> sp2 = sp1; // shared ownership

std::cout << sp1->denominator(); // access member
↪  function
```

To see how many shared pointers point to the same object, we can use the `use_count()` member function.

17.9 Memory Management and Garbage Collection

Memory management is not unique to C++. Languages like Java have a garbage collector that automatically frees memory that is no longer used. This has the advantage that the programmer does not need to worry about memory management. However, it also has the disadvantage that the programmer has less control over when memory is freed. In particular, this can lead to performance issues if the garbage collector runs at inopportune times. This makes C++ more suitable for time and memory critical applications.

18 Dynamic Data Structures II

The goal of this section is to create our own version of a vector, `our_vector`. To do this we will look at how to allocate whole blocks of memory, not just single objects.

18.1 Dynamic Arrays

We can use the `new` operator together with `[]` to allocate consecuitve chuncks of memory. The statement `new T[n]` allocates memory for `n` objects of type `T`. This chunck of memory is called an ARRAY.[2]

The `new T[]` operator returns a pointer to the first element of the array. But how do we access later elements? This is done using pointer arithmetic.

Pointers can be added or subtracted with integers. For a pointer `p + i` contains the address of the `i+1`-th element after `p`. The address `p+n` denotes the end of the array.

```
int* p0 = new int[5]{1,2,3,4,5};
int* p3 = p0 + 3;
*(p3 + 1) = 600; // sets p0[4] to 600
std::cout << *(p0 + 4); // outputs 600
```

Notice also the way to initialize arrays using the brace syntax.

Similarly, pointers can be subtracted. For example the expression `p2 - p1` gives the number of elements between the two pointers `p1` and `p2`. This leads to undefined behavior if the two pointers do not point into the same array or if `p2` is before `p1`.

As a warning, pointer arithmetic and integer arithmetic are not the same. Adding 1 to a pointer does not increase its value by 1, but by the size of the type it points to.

The SUBSCRIPT OPERATOR `[]` simplifies access to array elements. In particular, `p[i]` is equivalent to `*(p + i)` if `p` is a pointer.

To iterate over an array, we should in general use sequential loops, for example

```
char* p = new char[3]{'a','b','c'};
for (char* it = p ; it != p + 3; ++it) {
    std::cout << *it << std::endl;
}
```

This is faster than using indexing, as the programm then does not need to compute the address of each element again and again.

In C++ it is convention to pass arrays to functions using *two* pointers. In particular, one pointer to the beginning of the array and one pointer to the end of the array.

```
void fill(int* begin, int* end, int value){
    for (int* it = begin; p != end; ++it) {
        *it = value;
    }
}
```

[2] `n` has to be a non-negative integer expression.

18.2 Array-Based Vector

We can now try and build our own version of a vector, called `our_vector`.

```
class our_vector {
private:
    int* elements;
    int count; // number of elements
public:
    our_vector(int size);
    int size() const;
    int& operator[](int i);
};
```

The constructor creates an array of the given size and stores the size in `count`.

```
our_vector::our_vector(int size) : count(size),
↪  elements(new int[size]) {}
```

As a warning, this does not initialize the elements of the array.

The `size()` member function is particularly simple.

```
int our_vector::size() const {
    return count;
}
```

The index operator returns a reference to the `i`-th element. This is so we can both read and write elements.

```
int& our_vector::operator[](int i) {
    return elements[i];
}
```

If we also want to support reading from const vectors, we overload the index operator again.

```
const int& our_vector::operator[](int i) const {
    return elements[i];
}
```

Additionally, we should define an iterator for our vector. Within our class we already have an iterator (`int*`)[3]. Thus, we only need to define `begin()` and `end()` member functions.

```
class our_vector {
...
public:
    using iterator = int*;
    iterator begin() { return elements; }
    iterator end() { return elements + count; }
};
```

18.3 Memory Management with Dynamic Arrays

At the current moment in time, we allocate memory for the array in the constructor, but never free it. This leads to memory leaks. For every `new[]` there must be a matching `delete[]`. Notice that this does not set the corresponding pointer to `nullptr` but leaves it dangling.

[3]Remember, pointers are iterators!

Therefore, we need to define a destructor for our vector.

```
our_vector::~our_vector() {
    delete[] elements; // free array memory
}
```

But here the rule of three applies again! We also should define a copy constructor and assignment operator.

```
our_vector::our_vector(const our_vector& other)
    : count(other.count), elements(new int[other.count])
    ↪  {
    // copy elements
    for (int i = 0; i < count; ++i) {
        elements[i] = other.elements[i];
    }
}
our_vector& our_vector::operator=(const our_vector&
↪  other) {
    if (this != &other) { // self-assignment check
        our_vector copy = other; // copy constructor
        std::swap(count, copy.count);
        std::swap(elements, copy.elements);
    }
    return *this;
}
```

18.4 Insert and Remove Elements

An important feature of vectors is the ability to insert and remove elements. However, this is not possible with our current implementation, as we have a fixed size array. We want to give the general idea here.

A vector typically has a CAPACITY, which is the size of the allocated array, and a SIZE, which is the number of elements currently stored in the vector. When inserting an element, if the size equals the capacity, we need to allocate a new array with larger capacity, copy the elements over, and free the old array.

19 Object-Oriented Programming

Let us consider an artificial example of modelling cats and dogs from the real world. We have the following class for `Cat`:

```
class Cat {
    std::string name;
    int mice;
public:
    Cat(std::string name, int mice) : name(name), mice(mice) {}

    void greet(){
        std::cout << "Meow! I'm " << name << " and I caught " << mice << " mice!\n";
    }
}
```

We also want to model dogs now. They should also have a name and a number of postmen they have bitten. Similarly they should be able to greet us. To do this, we could copy the code from the `Cat` class and modify it slightly to achive the result.

The solution to this is to use INHERITANCE. We can define a base class `Animal` which contains the common features of cats and dogs.

```
class Animal {
    std::string name;

public:
    Animal(std::string name) : name(name) {}

    void greet(){
        std::cout << "I'm " << name << "!\n";
    }
};
```

We now can let `Cat` and `Dog` inherit from `Animal`.

```
class Cat : public Animal {
    int mice;
public:
    Cat(std::string name, int mice) : Animal(name), mice(mice) {}
};
```

We have to call the constructor of the base class in the constructor as `name` is private in `Animal`.

Sadly, we have lost the ability to greet with the number of mice caught. To fix this, we can OVERRIDE the `greet()` method in `Cat`.

```
class Cat : public Animal {
    int mice;
public:
    Cat(std::string name, int mice) : Animal(name), mice(mice) {}

    void greet() {
        Animal::greet(); // call base class greet
        std::cout << "I caught " << mice << " mice!\n";
    }
};
```

Using our implementation, we could alsp create an `animal` by calling `Animal a = Animal("Anna");`

Maybe surprisingly, it is also possible to create a `Cat` and store it in an `Animal` variable by `Animal a = Cat("Cathy", 5);`. In this case, the `greet()` method of `Animal` would be called, so we would not get information about the number of mice caught. This is called SLICING.

If we however use POINTERS or REFERENCES, slicing does not occur. This concept is called SUBTYPING.

Unfortunately, in our current implementation, calling the function `greet()` on an `Animal` pointer or reference to a `Cat` would still call the `Animal`'s `greet()` method. To fix this, we need to declare `greet()` as VIRTUAL in the base class.

```
class Animal {
    std::string name;
public:
    Animal(std::string name) : name(name) {}

    virtual void greet(){
        std::cout << "I'm " << name << "!\n";
    }
};
```

Notice that in this way, one pointer can point to objects of different types at different times. This is called POLYMORPHISM.

In this context, it is also crucial to make the destructor `virtual` in the base class. Otherwise, deleting an object through a base class pointer would lead to undefined behavior.

If we want to prevent a class from being created directly, we can declare it as an ABSTRACT CLASS by adding a PURE VIRTUAL function. A pure virtual function is declared by adding `= 0` at the end of its declaration.

```
class Animal {
    std::string name;
public:
    Animal(std::string name) : name(name) {}

    virtual void greet() = 0; // pure virtual function
};
```