Informatics

GianAndrea Müller

June 10, 2018

Contents

Terms

Algorithm An algorithm is a set of rules that defines a sequence of operations leading to the solution of a problem.

Language A programming language is a set of instructions for a computer that can be used to write programs that implement algorithms.

Syntax The syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured program in that language. Is a program grammatically correct?

Semantics The semantics of a computer language define how the language has to be interpreted. What is the meaning of a certain program?

Editor A program that allows writing code. There exist powerful editors that can check syntactical correctness on the fly.

Compiler A compiler translates a program written in a programming language to machine code, such that it can be executed by the machine.

Computer A computer is a device that is capable of executing machine code.

Comments Comments document the implemented algorithm within the program for the reader. They are ignored by the compiler.

Include Directives Include directives specify the additional libraries used in a program.

The main function The main function exists in every c++-program. It is unique and contains the all instructions necessary to execute the program.

Statement Statements are the building blocks of a program. They are executed sequentially and end with a semicolon.

Declaration A declaration introduces a new name to the program.

Definition A definition introduces a body to a name within the program.

Initialization An initialization introduces a value to a defined name.

Literals Literals represent constant values within the program. They have a defined type and value.

Variables Variables represent possibly changing values within the program. They have name, type, value and address.

Objects Objects represent values in the computer memory. They have type, adress and value. They can be named, but can also be anonymous. Described less generally an object can be a variable, a data structure, a function, or a method.

Expressions Expressions represent calculations. They are a combination of values, literals, operators and functions. They are primary if they consist of a single name/literal. Otherwise they are compound. They have type and value.

Lvalue An lvalue is a changeable expression that has an address.

Rvalue An rvalue is an expression that is not an lvalue. An rvalue cannot be changed. Every lvalue can be used as an rvalue but not vice-versa.

Operator An operator connects expressions to compound expressions. It specifies the expected operand(s) in type and if it is an rvalue or an lvalue. Operators have an arity.

Arity Arity is the number of arguments or operands an operator or a function takes. For example there exist unary (single argument) and binary (two arguments) operators.

Block A block in c++ is a number of lines of code enclosed by curly brackets.

1 Positional Notation

In short, a positional notation is defined as a method of representing or encoding numbers. It is characterized by its use of the same symbol for different orders of magnitude (for example the “ones place”, the “tens place” and the “hundreds place” in case of the decimal system). With the inclusion of the radix point, a symbol separating the integer part of the number of the fractional part, positional notation can represent rational numbers.

1.1 Binary Numbers

A binary number is a number expressed by a base-2 numeral system which uses only two symbols, typically 0 and 1.

1.1.1 From Decimal to Binary

To find the binary representation of a decimal number repeatedly divide by two and list the remainders. The resulting sequence is the binary representation in reverse order. For comparison the same algorithm is applied to a decimal number on the left.

$$
\begin{aligned}
91310 &= 10 * 9131 + & 0 \\
9131 &= 10 * 913 + & 1 \\
913 &= 10 * 91 + & 3 \\
91 &= 10 * 9 + & 1 \\
9 &= 10 * 0 + & 9
\end{aligned}
$$

$$
\begin{aligned}
61 &= 2 * 30 + & 1 \\
30 &= 2 * 15 + & 0 \\
15 &= 2 * 7 + & 1 \\
7 &= 2 * 3 + & 1 \\
3 &= 2 * 1 + & 1 \\
1 &= 2 * 0 + & 1
\end{aligned}
$$

1.1.2 From Binary to Decimal

To find the decimal representation of a binary number simply list its digits, multiply each with its place value and sum up.

							$\sum$
Digits	1	1	1	1	0	1	
Factor	32	16	8	4	2	1	
Value	32	16	8	4	0	1	61

1.1.3 Two's Complement

Two ways of interpreting signed numbers are shortly compared here in order to motivate the convention used.

Sign and magnitude

Bits	Unsigned value	S & M	Bits	Unsigned value	S & M
0000	0	0	1000	8	-0
0001	1	1	1001	9	-1
0010	2	2	1010	10	-2
0011	3	3	1011	11	-3
0100	4	4	1100	12	-4
0101	5	5	1101	13	-5
0110	6	6	1110	14	-6
0111	7	7	1111	15	-7

The interpretation with sign and magnitude interprets the first bit as the sign bit. This results in a range of $-(2^{N-1}-1)$ to $2^{N-1}-1$, where N is the number of available bits.

Two's complement

Bits	Unsigned value	2's	Bits	Unsigned value	2's
0000	0	0	1000	8	-8
0001	1	1	1001	9	-7
0010	2	2	1010	10	-6
0011	3	3	1011	11	-5
0100	4	4	1100	12	-4
0101	5	5	1101	13	-3
0110	6	6	1110	14	-2
0111	7	7	1111	15	-1

The two's complement interprets the first bit as the negative of its value in an unsigned interpretation. As seen above this allows omitting -0 as an encoded value and thus enlarges the range by one number: $-(2^{N-1})$ to $2^{N-1}-1$.

1.1.4 Overflow and Underflow

When calculating with numbers in a restricted range of values as on computer memory arithmetic operations can lead to over- and underflows. This is dangerous since there is no error message for such miscalculations.

unsigned int When an unsigned int is assigned a negative value (underflow) the resulting value can be described as follows:

$$\textbf{unsigned int } u \leftarrow x = \begin{cases} x \geq 0 & x \\ x < 0 & x + 2^B \end{cases}$$

Where 2^B is twice the value of the most significant bit and B is the number of bits. When the assigned value is in two's complement (which it is for c++) the representation does not have to be changed internally. Instead the unsigned interpretation effects the addition of 2^B since now the first bit of the number is not given a negative but a positive value. Note that the only case this does not happen is if the first bit is zero, which means that the assigned number was non-negative in the first place!

int When an int is assigned a value larger than can be saved a so called overflow happens, which means that the first bit of the number is switched and a large negative number results.

1.1.5 Estimation of the Order of Magnitude

$$
\begin{aligned}
2^{10} &= 1024 = 1Ki \approx 10^3 \\
2^{32} &= 3 \cdot (1024)^3 = 4Gi \\
2^{64} &= 16Ei \approx 16 \cdot 10^{18}
\end{aligned}
$$

1.2 Floating Point Numbers

A possible representation of numbers with a fractional part is fixing the number of digits before and after the radix point. The disadvantage of such a representation lies in its limitedness. A floating point system can tremendously increase the flexibility of such a notation. It consists of a certain number of significant digits and the position of the radix point. In other words it has a mantissa and an exponent. Note that no floating point system can fully represent $\mathbb{R}$, since the length of the mantissa is limited.

1.2.1 Description

Any floating point system can be described with a small set of parameters:

$$\mathcal{F}^{(*)}(\underbrace{\beta}_{\text{Basis} \geq 2}, \overbrace{p}^{\text{Length of mantissa} \geq 1}, \underbrace{e_{min}, e_{max}}_{\text{smallest and largest exponent}})$$

Where the asterisk, when present, signalises that the represented number always starts with a 1 (only works for basis 2). This first digit is called the hidden bit, since based on the above assumption, it does not need to be saved explicitly. Nevertheless it is counted as part of the mantissa.
When saved in computer memory the available bits are assigned in the following manner. Example for $\mathcal{F}^*(2, 6, -7, 7)$.

0000000000

- Sign bit
- Mantissa bits
- Exponent bits

1. All number are in normalized representation, which means that the exponent is chosen such, that there is a single 1 on the left side of the radix point.
2. The exponent is read as an unsigned int with a bias, for this example we shift the value range by 8, such that $0000 \hat{=} -8$.
3. For making certain numbers available that are otherwise not representable the smallest exponent 0000 is given up for encoding certain special values as shown below:

$$\underbrace{0000000000}_{\text{Null}} \quad \underbrace{0000000001}_{+\infty} \quad \underbrace{0000000010}_{-\infty} \quad \underbrace{0000000011}_{\text{NaN}}$$

1.2.2 Smallest Normalized and Largest Positive Number

$$2^{e_{min}} \qquad \left(1 - \left(\frac{1}{2}\right)^{p}\right) \beta^{e_{max}+1}$$

1.2.3 Decimal to Binary Representation

x	d_i	$x - d_i$
1.934	1	0.934
9.34	9	0.34
3.4	3	0.4
4	4	0

x	b_i	$x - b_i$
1.9	1	0.9
1.8	1	0.8
1.6	1	0.6
1.2	1	0.2
0.4	0	0.8
1.6	1	0.6
	⋮	

$1.1\overline{1100}$

What is concluded from this example that not all numbers are finitely representable and we can only save finite representations in the given floating point system.

1.2.4 Binary to Decimal Representation

									$\sum$
digit:	1	1	0	1	.	1	0	1	
factor:	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	
value	8	4	0	1		$\frac{1}{2}$	0	$\frac{1}{8}$	13.625

1.2.5 Guidelines when Using Floating Point Arithmetics

1. Single precision: $\mathcal{F}^*(2, 24, -126, 127)$
2. Double precision: $\mathcal{F}^*(2, 53, -1022, 1023)$
3. Never test rounded floating point numbers for equality!
4. Never add two floating point numbers of very different magnitude!
5. Never subtract two numbers of comparable size!

1.3 Hexadecimal Numbers

A hexadecimal number is a number expressed by a base-16 numeral system which uses the symbols 0-9 and A-F. For conversion between hexadecimal and decimal apply the algorithms learned for binary numbers. On preference convert to binary first, using hex nibbles.

1.3.1 Hex Nibbles

Any of the hexadecimal digits can be understood as a nibble (4 bits). Therefore hexadecimal numbers can be viewed as a compact representation of binary numbers, since every hexadecimal bit directly translates to a certain nibble as listed below.

hex	bin	dec	hex	bin	dec
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	A	1010	10
3	0011	3	B	1011	11
4	0100	4	C	1100	12
5	0101	5	D	1101	13
6	0110	6	E	1110	14
7	0111	7	F	1111	15

2 Extended Backus-Naur-Form

The Extended Backus-Naur-Form (EBNF) is a formal metalanguage used to describe context free grammars.

- **metalanguage:** A language about language.
- **context free grammar:** Rules for composing the words of a language that do not depend on the context.

Short and simple: The EBNF is a language that defines what sentences can be built with the words of another language, with a simple syntax. The EBNF consists of three main elements:

1. **Terminals:** Symbols that are the elemental basis of the language and cannot be further replaced.
2. **Nonterminals:** Symbols that can be replaced by other symbols based on certain rules.
3. **Derivation rules:** Rules that define which nonterminals are to be replaced by which terminals and in what manner.

The syntax for EBNF can be summed up as follows:

Usage	Notation	Usage	Notation
definition	`=`	grouping	`( ... )`
concatentation	`,`	terminal string	`" ... "`
termination	`;`	terminal string	`' ... '`
alternation	`|`	comment	`(* ... *)`
optional	`[ ... ]`	special sequence	`? ... ?`
repetition	`{ ... }`	exception	`-`

As an example we have here an EBNF defining whole numbers:

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6"
   | "7" | "8" | "9" ;
digit                = "0" | digit excluding zero ;
natural number = digit excluding zero, {digit};
whole number = "0" | [ "-" ], natural number;
```

A natural application of the EBNF is the definition of programming languages. The use of recursive functions then allows direct implementation of the defined derivation rules resulting in a parser. A parser is a program that checks whether a stream of symbols adheres to a certain grammar.

3 Character encoding

`char` represents printable characters as well as small set of control characters. Formally `char` is an integer type, mostly 8 bit large, encoded in ASCII code.

3.1 ASCII

ASCII coded uses the last 7 bits of char to encode most of the symbols used when typing text. The first bit has a special role.

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Practically the alphabet is stored in such a way that the last 5 binary digits indicate the position of the letter in the alphabet as seen in the figure below:

$$
\begin{aligned}
(65)- \quad & 10\ 00001 = & 'A' \\
(66)- \quad & 10\ 00010 = & 'B' \\
& \vdots & \\
(97)- \quad & 11\ 00001 = & 'a' \\
(98)- \quad & 11\ 00010 = & 'b'
\end{aligned}
$$

4 Syntax

4.1 Basic Program

```cpp
#include <iostream>
//#include "local_header_file.h"

/*
 * Function declarations (and definitions)
 */

int main(int argc, char ** argv)
{
    /*
     * Function calls
     */
    std::cout << "Hello World!" << std::endl;
    return 0;
}

/*
 * Function definitions
 */
```

4.2 Identifiers

A valid identifier, i.e. the name of a variable is:

- an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters.
- not starting with a digit.
- not starting with two or more underscores.
- not starting with an underscore followed by a capital letter.
- not a keyword of cpp.

4.3 Comments

C++ allows masking code such that it is not interpreted as part of the program. This enables documenting the program. There are different possibilities:

```cpp
// normal comment

/*
multi
line
comment
*/
```

Both versions can be nested:

```cpp
///*comment in a comment*/

/*
cout<<"Hello World!<<endl; //comment in a comment
*/
```

4.4 Data Types

4.4.1 Primitive Types

bool `bool` is used to save logical values. Check out section 7.3.

char `char` is used to represent characters and relies on encoding in ASCII. See section 3 for more detail.

int `int` is used to save integer numbers. For clarification check out section 1.

float & double `float` and `double` are used to save floating point numbers. For clarification check out section 1.2.

void The void data type cannot be used as a variable since it does not allocate any memory. However there exist a number of uses:

1. **Function return type:** A function with return type void does not return a value and does not need to have a return statement.

```cpp
void print(int a){
  std::cout<<a<<std::endl;
}
```

2. **Function parameter list:** A function with a `void` parameter list does not take any arguments.

```
void print(void){
  std::cout<<"I'm pretty lame."<<std::endl;
}
```

4.4.2 Type Modifiers

Modifier	Typical Bit Width	Typical Range
char	1byte	-127 to 127
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4byte	-2'147'483'648 to 2'147'483'647
unsigned int	4bytes	0 to 4'294'967'295
signed int	4bytes	-2'147'483'648 to 2'147'483'647
short int	2bytes	-32'768 to 32'767
unsigned short int	2bytes	0 to 65'535
signed short int	2bytes	-32'768 to 32'767
long int	4bytes	-2'147'483'648 to 2'147'483'647
signed long int	4bytes	-2'147'483'648 to 2'147'483'647
unsigned long int	4bytes	0 to 4'294'967'295
float	4bytes	+/- 3.4e +/- 38 (7 digits)
double	8bytes	+/- 1.7e +/- 308 (15 digits)
long double	8bytes	+/- 1.7e +/- 308 (15 digits)

4.4.3 Find minimum and maximum value of int

```
#include <iostream>
#include <limits>
using namespace std;

int main() {
   cout << "Minimum int value is "
        << numeric_limits<int>::min() << ".\n"
        << "Maximum int value is "
        << numeric_limits<int>::max() << ".\n";
   return 0;
}
```

4.4.4 Type Qualifiers: const volatile restrict

const Objects of type `const` cannot be changed by the program during execution.

const-correctness: Any variable that does not change its value during the course of the program is to be declared as const!

4.4.5 Variable Definition

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and name. The type has to be either a primitive type or a user defined object. Multiple names can be introduced at once if separated by commas.

```
int i,j,k;
```

This line both declares and defines the variables i,j and k. Direct initialization is also possible:

```
int d = 3, f = 2;
```

If a variable is left uninitialized its value is undefined in general. However, variables with static storage duration are implicitly set to 0.

4.4.6 typedef and using

You can create a new name for an existing type with `using` or `typedef`:

```
//typedef type newname;

typedef unsigned int uint;

//using newname = type;

using uchar = unsigned char;

uint a = 3; //a is an unsigned int

uchar b = '2'; //b is an unsigned char
```

5 Variable Scope

A scope is a region of the program within which variable definitions persist. There are three types of scopes:

1. Inside a function or a block (local variables)
2. In the definition of function parameters (formal parameters)
3. Outside of all functions (global parameters)

5.1 Blocks

A block is a group of statements enclose by curly brackets.

```
{statement1; statement2; ... statementN;}
```

Control statements generate blocks as well:

```
for (int i = 0; i < 10; i++){
  cout<<i<<endl;
}
// cout<<i<<endl; //not possible i is out of scope
```

5.2 Local Variables

Variables that are defined within a function or a block can only be used by statements within that same function or block.

```
for(int i = 0; i<3; i++){ //scope of the variable i
  std::cout<<i<<" ";
} //end of scope

// std::cout<<i; will result in error
```

6 Literals

6.1 Integer Literals

An integer literal can be a binary, decimal, octal or hexadecimal constant. A prefix specifies the base: `0b` for binary, `0` (zero) for octal, `0x` for hexadecimal and nothing for decimal.
An integer literal can also have a suffix that is a combination of u (unsigned) and l (long).

6.2 Floating-point Literals

A floating point literal has an integer part, a decimal point, a fractional part and an exponent part. You can represent floating point literals either in decimal form or exponential form. While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

6.3 Boolean Literals

There are two boolean literals and the are part of standard C++ keywords: `true` and `false`. The conversion from integer to bool results in `true` for all nonzero values and in `false` for zero.

6.4 Character Literals

Character literals are enclosed in single quotes (’). A character literal can be a plain character (’x’), an escape sequence (’`\t`’) or a universal character (’`\u02C0`’).

`\\`	\\ character	`\f`	Form feed
`\’`	’ character	`\n`	Newline
`\"`	” character	`\r`	Carriage return
`\?`	? character	`\t`	Horizontal tab
`\a`	Alert or bell	`\v`	Vertical tab
`\b`	Backspace		

6.5 String Literals

String literals are enclosed in double quotes. A string contains any combination of plain characters escape sequences and universal characters.

7 Operators

7.1 Arithmetic Operators

`+`	Adds two operands
`-`	Subtracts the second operand from the first
`*`	Multiplies both operands
`/`	Divides the first operand by the second
`%`	Returns the remainder of an integer division of the two operators
`++`	Increases an integer value by one
`--`	Decreases an integer value by one

- Note that the `/` operator returns a value of the same type as its operators. Thus a division of two integers, known as **integer division**, returns the rounded (down) result of the division.

7.1.1 Div-Mod Identity

`a/b * b + a%b == a`

7.1.2 Pre- and post-in-/decrement

The operators used for increment and decrement each have two tasks. The can be prefixed and postfixed and depending on this order have different effects. We will explore those effects considering the increment:

- **Pre-increment**: Increase the value of the argument and return a reference to the increased variable.
- **Post-increment**: Increse the value of the argument and return the previous value.

The signatures of those two functions are as follows:

```
//pre-increment outside class declaration
T & operator++(T& a);
//post-increment outside class declaration
T operator++(T& a, int);
```

From the signatures the difference in implementation that allows distinction between the two overloads becomes visible: The post-increment has a dummy parameter of type int. The selection of the overload happens in the background depending on the order of the call, no dummy parameter has to be handed over. Two possible implementations, making for the expected post condition and fitting the declarations above could be the following:

```
//pre-increment: return by reference
int& operator++(int& a){
  a = a + 1;
  return a;
}

//post-increment: return by value
int operator++(int& a, int i){
  int temp = a;
  a = a + 1;
  return temp;
}
```

7.2 Relational Operators

==	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

7.3 Logical Operators

&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand.

7.3.1 De Morgan's Laws

```
!(a && b) == (!a || !b)
!(a || b) == (!a && !b)
```

7.3.2 Application: XOR

XOR is the exclusive or operation and different descriptions can be derived using De Morgan's laws.

`(x||y) && !(x && y)`	x or y, and not both
`(x||y) && (!x || !y)`	x or y, and one not
`!(!x && !y) && !(x && y)`	not both and not none
`!(!x && !y || x && y)`	not: none or both

7.3.3 Short Circuit Evaluation

The logical operators && and || are left associative, thus evaluate the left operand first. If the result of the evaluation is clear after that, the right side is not evaluated at all.
The simpler evaluation should always be on the left of the operator.

7.4 Assignment Operators

=	Simple assignment operator, Assigns values from right side operands to left side operand.
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.

7.5 Misc Operators

`sizeof`	The sizeof operator returns the size of a variable in bytes. For example, sizeof(a), where 'a' is integer, and will return 4.
`?X:Y`	If condition is true then it returns value of X otherwise returns value of Y. Syntax: `<condition>?<then>:<else>`
`,`	The comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
`.`	The member operator `.` accesses a member of the operand on the left as specified by the operator on the right.
`->`	The member operator `->` is used to dereference the operand on the left and access one of its members indicated by the operand on the right.
Casts	Casting operators convert one data type to another. For example `int(2,200)` would return `2`.
`&`	The address operator returns the address of a variable.
`*`	The dereference operator returns the value the operand points to.

P.	Operator	Description	Associativity
1	`::`	Scope resolution	
2	`a++ a--`	Suffix/postfix increment and decrement	Left-to-right
	`type() type{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Locigal NOT and bitwise NOT	
	`(type)`	C-style cast	
	`*a`	Dereference	
	`&a`	Adress-of	
	`sizeof`	Size-of	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator	
9	`< <= > >=`	Relational operators	
10	`== !=`	Relational operators	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR	
13	`|`	Bitwise OR	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment	
	`+= -=`	Compound assignments	
	`*= /= %=`		
	`<<= >>=`		
	`&= ^= |=`		
17	`,`	Comma	Left-to-right

If written badly expression can result in undefined behaviour:

```
f(++i, ++i);
n = ++i + i;
b = ++a - a++;
```

Avoid changing variables which are used again in the same expression.

8 Conditional Statements

Conditional statements can be used to alter the flow of a program depending on set conditions.

8.1 if

The if command executes a single statement or a collection if a certain condition is evaluated as true.

```
if (condition) single_statement;

if (condition) {
  statement1;
  statement2;
}
```

8.1.1 if...else

The if else command executes statement1 if the condition is met and statement2 otherwise.

```
if (condition) {
  statement1;
}
else {
  statement2;
}
```

8.1.2 if...else if... else

Alternatively multiple conditions can be tested. Note that they are mutually exclusive, thus only the first of the cases, that evaluates as true is executed. The else if can be repeated.

```
if (condition1) {

}
else if (condition2) {

}
else {

}
```

8.2 switch

The switch statement is used to execute statements depending on the value of an expression. In doing so the expression is tested for equality with an integer (or enumeration type).

```
switch (expression) {
  case 1 : cout << '1'; // prints "1"
  case 2 : cout << '2'; // then prints "2"
}

switch (expression) {
  case 1 : cout << '1'; // prints "1"
    break;  // then exits the switch
  case 2 : cout << '2';
    break;
  default : cout << "default";
}
```

Note that the cases are only executed mutually exclusively if ended with the break command, that exits the switch statement. Otherwise, as in the first case above all consecutive cases after the first met case are executed until break or the end of the switch is encountered. The default case is executed anytime it is reached. In other words, if none of the above cases has been met or if one has been met an no break has been reached.

8.3 ? : Operator

The `? :` operator can be understood as a short form of an if else command.

```
a>b?a:b;

if(a>b) {
  return a;
}
else {
  return b;
}
```

The code snippets above accomplish exactly the same.

9 Loop Types

9.1 while

The while loop executes a set of statements as long as a set condition is evaluated as true.

```
while (condition) {
  statement;
}
```

9.2 for

The for loop not only tests a condition but has additional functionality which is normally used to instantiate a counter variable and increment it. The three together define the loop duration in one line.

```
for (init-statement; condition; expression){
  statement;
}

for (int i = 0; i < n ; i++){
  statement;
}

for (;condition;) {
  //equivalent to while(condition)
}
```

Optionally any of the three parts of the loop definition can be omitted. Leaving away the condition results in an infinite loop.

9.3 do...while

The do while loop executes its first iteration independent of the subsequently tested condition.

```
do {
  statement
} while(condition);
```

9.4 break continue

In all of the loops above the loop control statements can be used to escape the loop (`break`) to continue with the next iteration (`continue`).

```
while(true){
  if(condition){
    continue;
  }
  if(condition2){
    break;
  }
}
```

10 Functions

10.1 Structure

```
<r_type> <function_name> (<ptype> <pname>) {
  // body / definition of the function
}
```

- **Return type:** A function may return a value. The `return_type` is the data type of that value. If the function should not return a value its `return_type` is set to `void`.
- **Function Name:** The function name can be any valid cpp identifier and will be used, together with the parameter list to call the function. Together, name and parameter list are called the function signature, which is unique for every function.

  All non-void functions require a return statement!
- **Parameters:** The parameter variables are place-holders for the values that are passed to the function when it is called. On definition parameters must have type and name. Multiple parameters are separated with a comma. A function may have no parameters in which case a empty set of brackets is appended to the function name.
- **Function Body:** The function body contains a set of statements that define what the function actually does.

10.1.1 void

- `void` is a fundamental type that has an empty range of values.
- When used as a function return type void implies that the function does not return a value.
- void functions do not need a return statement.
- void functions end when encountering the end of the function body or when reaching the optional `return;`

10.2 Declaration and Definition

A function declaration informs the compiler that there is a function with a certain signature. The declaration consists of return type, function name and parameter list, which can omit the parameter names if not directly followed by the definition.

```
return_type function_name ( parameter list );

//for example
int max (int, int);
```

From that point on the function can be called within the code as long as it has a proper definition at some point. The definition can follow the declaration directly:

```
int min (int a, int b){
  return a<b?a:b;
}
```

Or be placed at some other point. Here we define the previously declared max function:

```
int max (int a, int b){
  return a>b?a:b;
}
```

10.2.1 Return Statement

The return statement concludes any function. When it is reached the function ends immediately, passing the returned value to the program calling it initially. The tpy eof the value given to the return statement has to match the return type defined in the declaration of the function.

10.3 Calling a function

A function is called by directly following the signature in the function declaration. For example the above defined max function can be called as follows:

```
int c = 2, d = 3;

cout<<"The maximum is "<<max(c,d)<<endl;
```

Upon this call the values c and d that are passed to the function initialize the two parameters a and b in the body of the function.

10.4 Function Arguments

Depending on the way the parameters are passed to the function it behaves fundamentally different.

10.4.1 Call by Value

A call by value copies the values handed to the function into the parameters of the function. Therefore changes made to the parameters inside the function do not have an effect on the argument.

```
void change(int a){
  a = 4;
}
```

10.4.2 Call by Reference

A call by reference makes the parameter a reference of the argument. Thus all changes made inside the function have the same effect on the argument.

```
void change_ref(int & a){
  a = 4;
}
```

The call does not differ to a call by value, the reference is made automatically:

```
int main(){
  int b = 3;
  change(b); //no effect
  change_ref(b); //b = 4;
}
```

Note that a call by reference can also be implemented using pointers:

```
void change_poi(int * a){
  *a = 4;
}
```

```
int main(){
  int b = 3;
  change(b); // no effect
  change_poi(&b); // b = 4;
}
```

Use call by read-only references instead of call by value for large data types to save effort.

10.4.3 Return by Reference

A function can also return a reference. This however is only possible if the function has been called by reference in the first place.

```
int & increment (int & i){
  i = i + 1;
  //pass reference to variable that exists outside
  return i;
}
```

When trying to pass a reference to a local variable there will be a runtime error.

```
int & increment (int no_reference){
  no_reference = no_reference + 1;
  //pass reference to local variable that
  //will not persist beyond the function scope
  return no_reference;
}
```

The motivation to return by reference lies in the idea of processing the return value of a function further, possibly with another function that has to be called by reference. A good example is a concatenation of assignments.

In contrast to a return by reference, using references, when using pointers we are not restricted to functions that were called by reference. It is possible to allocate new memory and return a pointer to that memory.

```
int * make_new_array (int Length){
  int * array_pointer;
  array_pointer = new int [Length];
  return array_pointer;
}
```

10.5 Recursion

It is possible to define function that call themselves:

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

In order to make the call above work we need **progress**, i.e. each function call needs to accomplish a step in the correct **direction**, i.e. the recursion has to head towards a certain goal, a **termination**. These are the three essential components of a reasonable recursion.

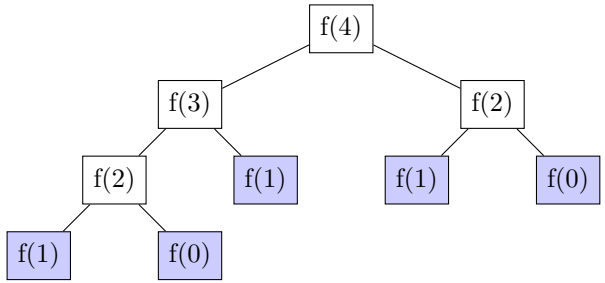
```
//POST: return value n!

unsigned int fac (unsigned int n)
{
  //termination condition
  if (n <= 1) return 1;

  //recursion with direction n->1
  return n * fac(n-1);
}
```

- Advantages:
  - Simple solution of complex problems
  - Easily understandable code
- Disadvantages:
  - Stack overflow possible
  - Slower
  - More difficult debugging

10.5.1 Call Stack

Recursive functions establish a whole stack of recursive calls until all ends are terminated. This stack can be nicely visualized:

```
//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
}
```



An iterative formulation of the above algorithm can also be established and nicely shows the advantage of its recursive brother:

```
// POST: return value is the n-th Fibonacci number F(n)
unsigned int fib2 (const unsigned int n) {
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1; // F_1
  unsigned int b = 1; // F_2
  for (unsigned int i = 3; i <= n; ++i) {
    unsigned int a_prev = a; // F_i-2
    a = b; // F_i-1
    b += a_prev; // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

11 Arrays

Arrays are used to save a previously known number of data points of the same type.

```
int list_1 [4]; // [r1 r2 r3 r4]
int list_2 [ ] = {1,2,3,4}; // [1 2 3 4]
int list_3 [4] = {1}; // [1 0 0 0]
list_1 [0] = 1;
//list_1 [4] = 5; // segmentation fault
```

- Arrays are declared with type, name and the length in square brackets. Uninitialized arrays contain random values. (`list_1`)
- Arrays with undefined length need to be initialized directly such that the length can be determined. (`list_2`)

- Arrays with a short initialization will fill up with zeros. (`list_3`)
- Array indexing starts with 0.
- Accessing out-of-bound values results in undefined behaviour.

11.1 Arrays and Pointers

The name of an array can be understood as a pointer to the first element of the array.

```
int list_1 [4] = {1,2,3,4};
int list_2 [1];
// list_1 = list_2; // error
int * list_pointer_1 = list_1;
int * list_pointer_2 = &list_1[0];
list_1[1]; // value: 2
list_pointer_1[1]; //value: 2
list_pointer_2[1]; //value: 2
*(list_1 + 1); //value: 2
```

- The name of an array is a pointer that is not allowed to be changed.
- A pointer to the name of an array is equivalent with a pointer to the address of the first element of the same array.
- The access operator `[]` works on the array name as well as on a pointer on the array.
- The access operator is equivalent with the increment and subsequent dereferencing of a pointer to the array.

Based on the knowledge that arrays are saved with a pointer to the first element of the allocated memory dynamic arrays can be created:

```
int n;
cin >> n;
int * dynamic_array = new int [n];
```

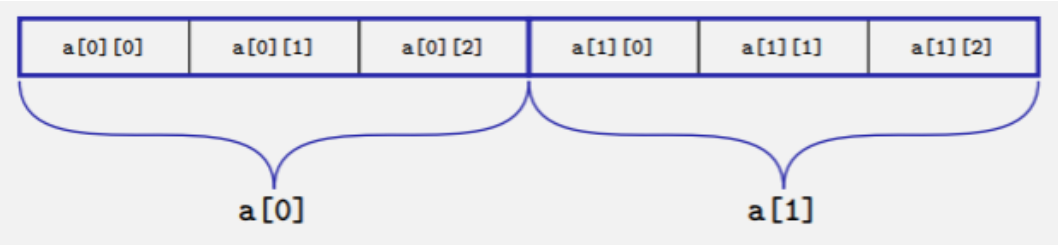
- The dynamically allocated array can now be used exactly the same as a static array.
- The dynamic array needs to be deleted with `delete [] dynamic_array` in order not to create memory leaks!

11.2 Multidimensional Arrays

Arrays can have more than one dimension, and can thus also be used to represent matrices or tables.

```
//2 elements, each an array of length 3
//uninitialized
int a_1[2][3];
//initialized, size specified
int a[2][3] = {
  {1,2,3},{2,6,3}
};
//initialized, size partially specified
int a[][3] = {
  {1,3,4},{2,3,4},{13 3 2}
};
```

As described in the last example it is possible to leave the first dimension unspecified if a direct initialization follows. Exactly as for simple arrays the length is determined automatically. This however does not work for the second dimension. The array above is saved in the memory as seen below:



11.3 Passing Arrays as Arguments

Since we know that arrays are basically pointers to memory it is possible to pass an array to the function by simple giving it a pointer to the first element of the array. Using references however we can prevent the decaying of the array-name (which remembers the length of the array if defined statically) by passing a reference to the array name as follows:

```
void some_array_function(int pointer []){
  //...
}

void nice_array_function(int (&arrayname) [10]{
  //...
}
```

The first function will accept any pointer and will not preserve the knowledge of the length

of the array. The second function is able to determine whether the argument is actually an array of the specified length and will preserve this information, but has the drawback that it only works for that specific length.

12 VECTOR

The vector class is a member of the standard library and improves on many of the issues observed with arrays. Vectors are implemented as a template and thus need a type specification in angled brackets.

```
#include <vector>

std::vector<int> int_vector;
```

The basic functionality of vector can be seen here, read for more details.

```
//create a bool vector with length 100
//all entries set to false
std::vector<bool> status(100,false);

//create a copy of an existing vector
std::vector<bool> copy(status);

//access entries
copy[0]; // first entry
copy.at(0); // first entry
copy.push_back(true); // append entry: true
copy.pop_back(); //delete last element
```

There also exists the possibility of filling up a vector with a certain value, over a certain interval, defined by two iterators.

```
#include <algorithm>

/*
std::fill(b,p,val) //where b,p are iterators
*/

std::vector<int> vec (5,4);             // vec: 4 4 4 4 4
std::fill(vec.begin()+3, vec.end(),2); // vec: 4 4 4 2 2
```

Another member function can find elements with a certain value within a vector and return an iterator to the first instance found. If nothing is found a past-the-end iterator (with respect to the defined interval) is returned.

```
#include <algorithm>

/*
std::find(b,p,val) // where b,p are iterators
*/

using It =std::vector<int>::iterator;
std::vector<int> vec = {8, 1, 0, -7, 7};
// Goal: Find index of -7 in vec: 8 1 0 -7 7
It pos itr = std::find(vec.begin(), vec.end(), -7);
std::cout << (pos itr - vec.begin()) << "\n"; // Output:
   3
```

Is is even possible to sort vectors from low to high.

```
#include <algorithm>

/*
std::sort(b,e) //where b,e are iterators
*/

std::vector<int> vec = {8, 1, 0, -7, 7};
std::sort(vec.begin(), vec.end()); // vec: -7 0 1 7 8
```

And we can find the minimum element of vector easily using:

```
#include <algorithm>

/*
std::min element(b, p) // where b,p are iterators
*/

// Goal: Make sure that all inputs are > 0
std::vector<int> vec;
int i;
while (std::cin >> i)
vec.push back(i);
assert( *std::min element(vec.begin(), vec.end()) > 0 );
// Note: We have to dereference the (r-value-)iterator.
```

Note that all the algorithms used above are made for use with the standard template libraries and thus work for other data structures in that library. Also this has only been a small selection of the available functions. See section 24 for additional information.

12.1 Multidimensional Vectors

Multidimensional vectors are similar to multidimensional arrays in functionality. The declaration is a bit more complicated:

```
using namespace std;
vector < vector < int > > a (n, vector<int>(m));
```

The above declaration describes a vector of int vectors. The initialization defines the dimensions as follows: The outer vector is of length n and contains vectors of lengths m. Note that the space between `< <` is necessary, since without it the brackets would be interpreted as an operator.

12.2 Access

There exist different methods of accessing the elements of vectors.

1. **Random access:** The concept of random access allows accessing any element of a vector with the same effort.

   ```
   //long array (syntax simplified!)
   int a [] = {1,2,3,4,5,6,...,100};

   //random access:
   a[33]; // identical to: *(a+33);
   ```

   The random access operation as described above requires a single addition and an implicit multiplication automatically made through pointer arithmetic when adding a number to a pointer.

   This access method is really flexible but also costly.

2. **Sequential access:** This is where sequential access shines, which only requires a simple addition for getting to the next element in an array, but is less flexible since it can only go back an forth in steps of one. Sequential access can be implemented as follows:

   ```
   int a[5] = {1,2,3,4,5};
   for (int* p = a; p < a+5; ++P)
     cout<<*p<<" ";

   //For vectors iterators can be used
   ```

12.3 Iterators

Since vectors allocate memory differently than arrays they cannot simply be traversed using a pointer. For that reason and to have an interface for advanced functionality vectors have iterators pointing to their elements.

```
std::vector<int>::const_iterator

std::vector<int>::iterator
```

The first version can be used for non-mutating access. It is analogue to a `const int*` for arrays. The second version allows changing of the elements it points to.

```
std::vector<int> vec;
vec.push_back(1);
vec.push_back(2);
std::vector<int>::iterator it = vec.begin();

while(it<vec.end()) {
  cout<<*it<<endl;
  it++;
}
```

In the above example the member functions `begin()` and `end()` are used to initialize and constrain an iterator going through the vector. Access to pointed-to elements and increment of the iterator works just like it does using pointers.

12.4 Passing Vectors to Functions

By convention of the standard library vectors are passed to functions using two iterators, one to the beginning of a valid part of the vector to be passed and a second to the ending of the same. This can of course include the whole vector but includes the flexibility of only processing a part of it.

```
std::vector<double> d(100,0);
std::fill(d.begin(),d.end(),1);
```

13 Strings

13.1 C-style Character String

One possibility to save a string is using an array of chars. By convention character arrays are terminated with `\0`. This serves the purpose of marking the end of a string.

```
//initialisation with string literal
char text = "bool";
//equivalent to:
char text = {'b','o','o','l','\0'};
```

13.2 string

The modern alternative to char arrays is the string class in the standard library. It serves the same purpose as the character array but makes handling strings a lot easier. For instance the length of a string does not have to be known at compiletime and can thus also change during runtime.

```
#include <string>
std::string text1 = "bool";
text1.length(); //A string knows its length
//initialize with variable length n
//and fill with 'a'
std:string text2 (n, 'a');
//string understands comparisons
//and many other operations
text1 == text2;
```

14 Pointers

Pointers are variables saving addresses of other variables.

```
int a = 6;
//b is a pointer to an int
//and takes the address of a
int * b = &a;

(*b)++; //a == 7
```

There are three important operators for pointers:

- Declaration of a pointer:

```
//<type> * <name>;
int * pointer;
```

- Adress of a variable:

```
//& <variable>;

int var;
pointer = & var;
```

- Accessing the value pointed to (dereferencing)

```
//want to set var = 7;
//*<initialised_pointer>;
*pointer = 7; //var == 7
```

14.1 Pointer Arithmetic

In connection with arrays there are more operations possible:

```
int my_array [10];
//Pointer to first element
int my_pointer = my_array;

while(my_pointer<my_array+10){
  //do something
  //increment pointer
  my_pointer++;
}
```

As seen in the example above it is possible to do calculations with pointers. This works since the arithmetic operators (`+,-,++,--`) are overloaded to work with pointers. Also relational operators (`<,>,<=,>=,==,!=`) can be directly used.
What happens internally is that the length of the data element pointed to is known by the type of the pointer and thus an increment leads to the desired change in the saved address, such that `my_pointer` points to one element in the array after the other.

14.2 Const Pointers

- A const pointer cannot point to another variable once initialized.
- A pointer to a const variable cannot change the variable it points to.
- A nonconst pointer cannot point to a const variable.

Read from right to left! Read * as 'Pointer to'.

- `int * iptr = &i;`
  iptr: Pointer auf int.
- `const int* icptr = &i;`
  icptr: Pointer auf const int.
- `int const * ic2ptr = &i;`
  icptr2: Pointer auf const int.
- `int * const iptrc = &i;`
  iptrc: const Pointer auf int.
- `const int * const icptrc = &i;`
  icptrc: const Pointer auf const int.

14.3 Additional Information

Also see section 11.1 for the connection between pointers and arrays, as well as 10.4.2 to see how pointers can be used to implement call by reference.
Also consider the additional information explaining many advanced uses of pointers.

14.4 Differences Between Pointers And References

- A pointer can change where it points to.
- A pointer can be uninitialized, thus point to `NULL`.
- A pointer has an address itself.
- There is no such thing as reference arithmetic.

A reference can be understood as a constant pointer that does not need to be dereferenced.

15 References

A reference is another name for an already existing variable. It has to be initialized when declared and cannot refer to another variable afterwards. **The object referred to has to exist at least as long as its reference!**

```
int i = 1;
int & ref_to_i = i;
// int & ref_to_nothing; //not allowed!
```

In c++ a reference declaration consists of the type which is to referred to, the `operator&` and the name of the reference, followed by an assignment of a valid L-value. After initialisation `ref_to_i` can be used exactly the same as `i`!

Only `const` references are allowed to refer to R-values!

```
//int & h = 3; // not allowed!
const int & i = 7;
```

Only `const` references can refer to constant variables.

```
const int n = 5;
// int & i = n; // not allowed
const int & i = n; // allowed, read only
```

16 Input/Output

C++ I/O occurs in streams, which are sequences of bytes. When bytes flow from a device like a keyboard, a disk drive or a file to main memory this is called an input operation. If bytes flow the other direction, which means from the program to a device like a display screen or a file this is called an output operation.

The most important libraries for basic IO are:

- `<iostream>`

  This file defines `cin, cout, cerr` and `clog`, which correspond to the standard input, output, unbuffered error and buffered standard error stream.

- `fstream`

  This file declares services for user-controlled file processing.

- `sstream`

16.1 cout and cin

```
#include <iostream>

//allows omitting std:: before cout and cin
using namespace std;

int main(){
  //declare variable
  int a;
  //initialise via keyboard input
  cin>>a;
  //print via console
  cout<<a;
}
```

These two streams can also be handed to functions which further allows overloading of `<<` and `>>`, such that for example more elaborate classes can be handed to `cout` directly. In the example below a suitable overload for a class implementing rational numbers is made:

```
//Stream operator
//PRE: Takes a ostream as an argument
//POST: Prints to the handed ostream
std::ostream& operator<<(std::ostream& os, const Rational
    & rational)
{
    os << rational.numerator << "/" << rational.
       denominator;
    return os;
}
```

It is important to notice that **the standard input stream and the standard output stream are unique objects that cannot be copied**! For that reason when handed to an operator as seen above they are handed as references! For an explanation why this operator returns a reference, see section 21.2.2.

16.2 ifstream and ofstream

These two streams allow reading from a file (`ifstream`) and writing to a file (`ofstream`). In the subsequent subsections we will build up all the code needed for the above described operations. All snippets contribute to the same program and can be run as a whole.

16.2.1 Opening a File

First we open a file in the desired mode using the `fstream` member `open()` Selectable file modes are:

- `ios::app`

  Append mode. All output to that file to be appended to the end.

- `ios::ate`

  Open a file for output and move the read/write control to the end of the file.

- `ios::in`

  Open a file for reading.

- `ios::out`

  Open a file for writing.

- `ios::trunc`

  If the file already exists, its contents will be truncated before opening the file.

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
  //declare an instance of ifstream
  ifstream infile;
  //open a file
  infile.open("file_name.txt", ios::in);
```

16.2.2 Reading From a File

After having set up the ifstream we can use it just like `cin` to read information from the file to main memory:

```
  char in;
  string whole_file;
  //extract from file until
  //end of file is reached
  while(!infile.eof()){
    //read next char
    infile>>in;
    //save char in string
    whole_file.push_back(in);
  }
```

16.2.3 Closing a File

After reading the file we will close it since it is of no further use.

```
  infile.close();
```

16.2.4 Writing to a File

Now we'd like to write our read and processed file content to another file. First we open a file.

```
  //Declare instance of ofstream
  ofstream outfile;
  //Open a file
  outfile.open("new_textfile.txt",ios::out);
```

The we use `outfile` to write to a new file.

```
  //After opening writing is possible
  while(!whole_file.empty()){
    char temp = whole_file.pop_back();
    //Some kinda caesar code
    temp++;
    //Write to file
    outfile<<temp;
  }
```

After that the file is closed.

```
  outfile.close();
  return 0;
}
```

16.3 sstream

String-Stream is a stream class that can handle strings. Same as the standard in-/out-streams instances of sstream cannot be copied and should be handed by reference. An exemplary use would be emulating cin-input:

```
char c;
std::cin >> c; // e.g. value: 'b'
// stringstream with value "b345e"
std::stringstream s ("b345e");
// c gets value 'b' and s is now "345e"
s >> c;
// c gets value '3' (!as char! since type of c is char)
s >> c;
int n;
// n gets value 45 (!as int! since type of n is int)
s >> n;
// (This works since the computer sees that the next
// 2 characters in the string "45e", namely '4' and '5',
// can be used as the int 45. So after this operation
// s is "e".)
```

16.4 Functionality of Streams

A stream can have different statuses and there are functions to check the status of a stream:

```
/*
//true if the end of file has been reached
my_stream.eof()
*/

std::stringstream b ("33");
int z;
b >> z; std::cout << b.eof();
// true (>> read 33 and reached end of b)
```

```
/*
//true if an invalid character has been read
my_stream.fail()
*/

std::stringstream b ("33a");
int z;
// false (>> read 33)
b >> z; std::cout << b.fail();
// true (>> ought to read 'a'
b >> z; std::cout << b.fail();
// which is not an int.)
```

```
/*
//return the next character converted to int
//without extracting it from the stream
my_stream.peek()
*/

std::stringstream num ("3 a");
// Output: 51 and NOT '3'
std::cout << num.peek() << "\n";
num >> c;
std::cout << c << "\n"; // Output: '3'
char next = num.peek();
std::cout << next << "\n"; // Output: ' '
num >> c;
std::cout << c << "\n"; // Output: 'a'
```

```
/*
std::ws //extract whitespaces until next character
std::noskipws //prevent auto-ignoring of whitespaces
*/

char c;
std::stringstream t ("d a\n\nb");
t >> std::noskipws; // Do not ignore whitespaces.
// Output t without whitespaces
t >> c; std::cout << c; // Output: 'd'
t >> std::ws; // Remove: " "
t >> c; std::cout << c; // Output: 'a'
t >> std::ws; // Remove: "\n\n"
t >> c; std::cout << c; // Output: 'b'
// Output in total: dab
```

17 STRUCT

Struct allows the definition of new custom data types.

```
/*
struct <name> {
  <type> <member_name>;
  .
  .
  .
}<optional_instance_name>;

*/

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

rational b;
```

- Now the new type `rational` is defined.
- It can be directly instantiated by inserting one or several (separated with comma) valid identifiers at the end of the struct definition. In this case we made instance `a`.
- When making new instances afterwards we can use the name of the new type just as we use the basic types.
- Member variables can be of basic type as well as of custom type, i.e. structures can be nested.
- Member types can be combined in any way desired.

Members can be initialized directly:

```
//uninitialized
rational s;

//member-by-member
rational t = {1,5};

//copy member-by-member
rational u = t;
```

Note that for custom data types not all operators work out of the box. This leads to the idea of overloading operators, as described in section 21.2.

18 CLASS

Classes empower the concept of data encapsulation in c++. They are a variant of structs and as such can have member variables as well as member functions.

Note that structs basically have the same functionality as classes. They are introduced differently for didactic reasons but actually the single difference between the two is that the members of a struct are public by default and the members of a class a private by default.

18.1 Object Oriented Programming

The basic idea of object oriented programming is to build software such that it models reality. For that reason the desired functionality is embedded into a set of objects interacting with one another to solve a set task. With object oriented programming come two main advantages:

- **Reusability:** Once described the functionality of a class can be instantiated multiple times, inherited from and adapted for the whole inheritance tree in a single place.
- **Encapsulation:** Based on the fact that variables can be made private it is possible to hide information from the user or from other parts of the program. This enables a narrow definition of allowed interactions with data, such that invalid access can be prevented by design.

18.2 Basics

- **Class:** The definition of an object, introducing a new data type.
- **Instance:** Existing copy of the newly defined data type.

18.3 Class Members

There are two types of class members: **member functions** and **member variables**. Member variables are data points within the class. Member functions can access these data points for processing. The members of a class exist in instances of the class.

```
/*
class <class_name> {
  //member variable
  <type> <member_name>;
  //member function
  <rtype> <member_fun_name> ();

}<optional_instance_name>;
*/
class Vector {
  //member defintion
}a;
```

As seen above instances of a class can be directly added to the class definition.

```
int main(){
  Vector b;
}
```

Alternatively the class name can be used as a type specifier, for introducing a new instance.

18.3.1 Member Functions

The declaration of a member function always has to take place within the class definition. However it is possible to separate declaration and definition and thus the definition can be placed outside the class definition.

```
class Vector {
public:
  void print();
  void explode(){
    cout<<"BOOM!"<<endl;
  }
};

Vector::print(){
  cout<<"Don't mind me, I'm just printing."<<endl;
}
```

The external definition needs to know that is is a member function of the class. This is done using `Vector::`.

Note that member functions can be declared const, which means that they cannot change any of the members unless they are mutable (see section **??**).

```
class Vector {
public:
  void doNothing() const { //nothing }
};
```

The const after the parameter list of the function is refering to the implicit `*this` argument (see section 18.4.3) which is accessed by every member function of class.

18.4 Class Access Modifiers

Class members, not yet considering inheritance, can be either private or public.

- Public members are accessible from outside.
- Private members are only accessible through member functions.

If there is no access modifier all members of a class are private by default!

```
class Vector {
  //default: private
  double x; //is private
private:
  double y; //is private
public:
  void print(); //is public
};
```

18.4.1 Access Methods

In order to define an interface to the hidden data one often defines get- and set-functions:

```
  //within definition of vector
  double get_x() const {return x;}
  double get_y() const {return y;}

  void set_x(const double _x) {x = _x;}
  void set_y(const double _y) {y = _y;}
```

18.4.2 Access Operators

When accessing the members of a class not from within the class but from the outside, through an instance, there exist two operators managing access to public members:

```
//Instance of Vector
Vector a;
//Pointer of type Vector to a
Vector * pointer = &a;

//Direct access operator .
a.set_x(7);

//Indirect access operator ->
pointer->set_x(7);
```

The `operator.` is used in connection with an instance of a class. It directly accesses the appended member. The `operator->` is used for accessing the members of an instance through a pointer. In addition to member access it dereferences the pointer.

18.4.3 This Pointer

Sometimes it is useful to have access to the instance of a class as an object, not only to its members. In that case the `this` pointer comes in handy. It is a pointer saving the address of the instance it is used in. Thus the above defined access methods could also look like this:

```
void get_x() const {return this->x;}
void get_y() const {return (*this).y;}
```

18.5 Constructors

- Standard member function, exists in default form if not overwritten.
- Used to initialize member variables.
- Automatically called upon instantiation.
- Carries the name of the class.
- Can be overloaded for different types of initializations.
- The default constructor does not initialize anything.

```
class Vector {
  double x,y;
public:
  //Overwriting the default constructor
  Vector (){
    x = 0;
    y = 0;
  }
  //Overloading the constructor
  Vector (double _x, double_y)
    : x(_x),y(_y) {}
}
```

The above implementation of the constructor overload makes use of the so called **initializer list** which is a list of member variables and their respective value in brackets. This comes in especially handy if a class has const members that cannot be initialized within the function body.

18.5.1 Copy Constructor

The copy constructor is used for:

- Initializing one object from another of the same type.
- Copying an object to pass it as an argument to a function.
- Copying an object to return it from a function.

The default copy constructor simply sets the member variables of the target equal with those of the source. This is not sufficient if a class has dynamic memory allocated.

```
/*
<classname> (const <classname> & obj) {
  //body of constructor
}
*/

Vector (const Vector& obj){
  this->x = obj.x;
  this->y = obj.y;
}
```

18.6 Destructor

- Standard member function, exists in default form if not overwritten.
- Used to deallocate memory taken by an instance.
- Is called whenever an instance is explicitly deleted or reaches end of scope.
- The destructor only needs to be overwritten in case the class allocates dynamic memory.

```
/*
~<classname> (){
  delete <dynamic_variable>;
  delete [] <dynamic_array>;
}
*/

~Vector (){
  cout<<"Vector is being destructed."<<endl;
}
```

For the lack of dynamically allocated memory the default constructor of Vector has been overloaded with this simple output, an easy way to see when the destructor is called.

18.7 friend

To allow a non-member function full access to the private variables of a class, friend functions can be defined as follows:

```
//outside class
void foreign_print(Vector obj){
  cout<<obj.x<<" "<<obj.y;
}

//inside class

friend void foreign_print(Vector obj);
```

18.8 Static Members

A static member of a class exists only once, independent of how many instances of that class are around. All instances work with one and the same static variable.

```
static <member_type> <member_name>;
```

19 Dynamic Data Structures

To illustrate the consequences of dynamic memory allocation in a custom data type, the example of a stack is shortly covered.

19.1 Linked List

The idea of a linked list is to enhance the capability of a simple array with more flexibility. We want to have the freedom to:

- Insert/Append data points
- Remove data points

This is achieved using so called linked lists, which in their simplest form are based on the following struct, in this case built for integer data points.

```
struct node {
  int datum;
  node * next;
}
```

Now the idea of a linked list:

1. Start the list with your first data point. Let its member `next` point to `NULL`. This is the end of our list.
2. Append a new data point by allocating it dynamically and setting its `next` pointer to the address of the old starting point of our list.

3. Always save a pointer to the start of the list. This is all the information needed to access all points in the list.

4. Traverse through the list by subsequently accessing the `next` pointers.

This is the idea of a singly linked list (linked in one direction) that can be used to implement a stack. The concept however allows much more.

19.2 Stack

The stack is a so called FIFO (First in first out) data structure. It can be associated with a stack of bricks. The one laid on top has to be removed again first.

```
class stack {
public:
  void push (int value);
  int pop ();
  void print();
private:
  node* top_node;
}
```

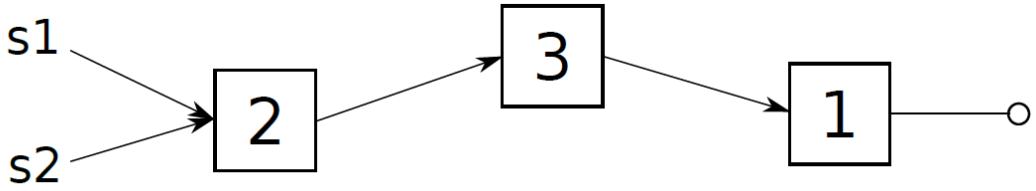
Our stack can now be used as follows:

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

19.3 Copy Constructor

Now if we use the standard copy constructor for our stack, our only member variable is copied, such that now we have two stacks using the same set of nodes:

```
stack s2(s1);
```



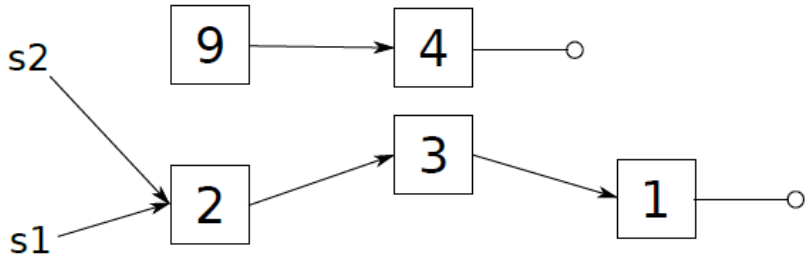
The issue is that if now `s1` pops one of the nodes `s2` will be pointing to a now inexistent node. In other words the two stacks are not independent. To resolve this the copy constructor is overwritten as follows:

```
stack::stack(const stack& s) : top_node (0) {
  copy(s.top_node, top_node);
}

void stack::copy(const node* from, node*& to) {
  assert (to == 0);
  if (from != 0) {
    to = new node(from->datum);
    copy(from->next, to->next);
  }
}
```

19.4 Assignment operator

Now assume we'd like to assign one stack to another:

```
stack s2;
s2.push(4);
s2.push(9);
s2 = s1;
```



The problems arising here are:

- We have two non-independent stacks againg.
- The nodes 9 and 4 of former `s2` are memory leaks.

To solve this the assignment operator is overloaded. First we need a function cleaning up memory leaks:

```
void stack::clear(node* from) {
  if (from != 0) {
    clear (from->next);
    delete from;
  }
}
```

```
stack& stack::operator= (const stack& s) {
  if (top_node != s.top_node) { // test for self-
      assignment
    clear(top_node);
    top_node = 0; // fix dangling pointer
    copy(s.top_node, top_node);
  }
  return *this;
}
```

19.5 Destructor

The final issue that needs to be addressed is the case when a stack is at the end of its scope and thus deleted. The standard destructor will just delete all member variables, and thus not clean up our dynamically allocated nodes. To solve this the destructor is overwritten as well:

```
stack::~stack() {
  clear(top_node);
}
```

19.6 Rule of three

If a class defines one or more of the following it should probably define all of them:

- destructor
- copy constructor
- copy assignment operator

All three functions above are compiler generated, thus exist automatically in their naïve default version. If one of them has been overwritten it means that the compiler made version does not suit the need of the underlying class. Thus it is very likely that the other two are not suitable as well.

20 Inheritance

The concept of inheritance allows building a class upon another, such that the basis inherits all its members to a derived class. This allows reusing code and reduces the amount of code written if multiple similar objects are needed. Further, if the basic traits of a family of objects need to be changed, if implemented correctly only the base class needs to be adapted.

```
class A{  //base class
  //...
};

//derived class
class B : public A {
  //...
};
```

20.1 Access Control and Inheritance

As classes have members with different access specifications there needs to be a way to define which member is available in what way in a derived class. This is done with the table below. The access specifier of the inherited member is on the horizontal direction. The type of inheritance is on the vertical direction.

Inheritance \ Member	`public`	`protected`	`private`
`public`	`public`	`protected`	n/a
`protected`	`protected`	`protected`	n/a
`private`	`private`	`private`	n/a

The type of inheritance is incorporated in the definition of the derived class as follows:

```
class base {
private:
  int pr_member;
public:
  int pu_member;
};

/*
class <derived_name> : <access modifier> <base_name> {
  //...
};
*/

class derived : private base {
  // pr_member not available
  // pu_member available as private
};
```

20.2 Constructors

Since private member variables are not accessible in derived classes, they need to be initialized differently:

```
class base{
  int Bvar;
public:
  base(int _Bvar) : Bvar(_Bvar){}
};

class deri = public base {
  int Dvar;
public:
  deri(int _Bvar, int _Dvar) : base(_Bvar), Dvar(_Dvar)
     {}
};
```

20.3 is a - has a

Classes can inherit from each other or they can contain each other as member variables.

```
class University {
private:
  std::vector<Student> students_;
};

class Student {
private:
  Legi legi_;
};

class Phys_Student : public Student {};

class Legi {
  int immatriculation_year_;
};
```

The above set of classes show the difference. A university **has** students. A physics student **is** a student. In implementation both versions can be used and yield very similar results. The decision which to choose is often based on the reality that is to be represented.

20.4 Polymorphism

Polymorphism is a concept based on the possibility to let pointers of the type of the base class point to derived classes.

```
class base {
  //...
};

class derived : public base {
  //...
};

derived derived_instance;
base * pointer = &derived_instance;
```

When now this pointer is used to call a member function defined in the base class and overwritten in the derived class, the decision which version to use needs to be made. Normally the member function, that is associated with the type of the pointer (thus base class) will be executed.

20.4.1 virtual

This behaviour can be changed when a function is declared as virtual in the base class. Then a pointer of the base class, pointing to a derived class, will use the function definition within the derived class. This is called **dynamic binding**.

```
class A {
  virtual void print()
  {cout<<"A"<<endl;}
};

class B : public A {
  void print()
  {cout<<"B"<<endl;}
};

A instantce1;
instance1.print(); // "A"
B instance2;
instance2.print(); // "B"
A * pointer1 = &instance2;
pointer1->print(); // "B"
```

This behaviour can be summed up as follows:

1. When called directly, via an instance of a class, the member function corresponding to the type of the calling instance is used.

2. When a non-virtual member function is called indirectly, via a pointer to an instance, the function corresponding to the type of the calling pointer is used.

3. When a virtual member function is called indirectly, via a pointer an instance, the function corresponding to the type of the calling instance is used.

21 Overloading

Functions and operators can work differently, depending on their arguments or operands.

21.1 Function Overloading

Functions overloading means varying the types, the order and the number of arguments given to the function. For each variation a different behaviour can be programmed. Upon call the handed arguments are compared with the overload signatures and the matching function is executed. If no matching function is found overload resolution rules are applied.

```
//a way of identifying unknown variable types:

void identify (int a){
  cout<<"My type is int."<<endl;
}

void identify (double a){
  cout<<"My type is double."<<endl;
}
```

Based on the rule stated above the following overloads are possible:

```
void fun();      //without argument
void fun(int a);  //with argument
void fun(double b); //different type
//different number of arguments
void fun(int a, int b);
//different type of arguments
void fun(int a, double b);
//different order of arguments
void fun(double b, int a);
```

Based on the above the following overloads are not valid:

```
//No distinction by return type
int fun();
//No distinction by argument name
int fun(int different);
```

21.2 Operator overloading

Operators are can be described like functions. Depending on their definition they have a set of arguments, i.e. left-hand side and right-hand side for `operator+` and a return value connected to a post condition, which in this case would be: returns the sum of the two handed variables. As done above, in c++ operators can be addressed via their name and the keyword `operator` in front.

Based on that idea it is obvious that overloads are also possible for operators, which allows defining them also for our custom data types. Thus an operator overload is defined as a normal function, using `operator<name>` as a function name, and defining the argument list such that it fits the chosen operator.

There is a binary `operator-` and a unary `operator-`. To show how they differ in their signature the following example is established for the custom data type `rational`.

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

The overloading of the binary operator, taking the value on its left and the value on its right:

```
rational operator- (const rational& l, const rational& r)
   {
  rational result;
  result.n = l.n*b.r - l.d*r.n;
  result.d = l.d*b.r;
  return result;
}
```

The overloading of the unary `operator-`, taking the value on its right.

```
rational operator- (const rational& r){
    rational result = r;
  result.n = -result.n;
  return result;
}
```

Note that the parameters are handed as const references, therefore they are not changed by accident and rather than copying the whole structure a referenced is handed to the function.

Both operators in use:

```
int main(){
  rational a = {1,3};
  rational b = {2,4};
  //unary operator-
  rational c = -b; // -2/4
  //binary operator-
  rational d = a-b;
}
```

21.2.1 Overloadable Operators

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new[]	delete	delete[]

Note that there are certain operators that cannot be overloaded:

::	.*	.	?:

Find examples for the overloading of any of the operator types listed above here

21.2.2 Chained operators

Certain operators allow to be chained together. Good examples are the assignment operator or the I/O operators << and >>. To preserve the ability to be put in chains, these operators need to return references. Why this is necessary will become obvious in the following example:

```
#include <iostream>

using namespace std;

cout<<"Hello"<<" world"<<'!'<<endl;
```

The `operator<<` is left-to-right associative, therefore its first evaluated occurrence is `cout<<"Hello"`. This term has to return a reference to the standard output stream such that the next evaluation `cout<<" world"` can be made properly.

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

22 Dynamic Memory

Normally memory is allocated according to the variables specified, before program execution. However it is often practical to allocate memory during runtime.

- Every `new` needs a fitting `delete`.
- Not cleaning up leads to memory leaks which can finally fill up the heap (heap overflow).

22.1 new

```
/*
  new <type> (<constructor args>);
*/

//allocation of a single variable
int * pointer;
pointer = new int;

//allocation of a whole array
//length does not have to be defined
//at compiletime!
int * another_pointer;
another pointer = new int [length];
```

- `new` allocates a new object of the desired type, calls the correct constructor as specified in the brackets and returns a pointer to the newly generated object. If no constructor is needed the brackets can be omitted.

22.2 delete

Object that are allocated using `new` have dynamic storage duration, the memory occupied by them is only released again if explicitly deleted.

```
/*
  delete <pointer>
*/

//delete the objects allocated above
delete pointer;

//Use delete [] for dynamically allocated arrays
delete [] another_pointer;
```

- `delete` removes dynamically allocated objects.
- `delete` needs a pointer to the object in order to delete it.

23 Namespaces

- Different libraries using the same function name call for a distinction.
- `namespace` allows wrapping declarations in an envelope that distinguishes them from any definition with the same name.
- With `using namespace <namespace_name>` the compiler is informed that functions from that namespace are used. This does not work if we introduce ambiguities:
- If we were `using namespace n1;` in the example below, there would be an ambiguous function overload, which leads to a compilation error.
- Namespaces can be discontiguous, i.e. split into different parts in different files. If part of the namespace requires a name defined in another file, that name must still be declared!
- There can be namespaces inside namespaces.

```
/*
namespace <namespace_name> {
  // code declarations
}
*/

namespace n1 {
  void myfun(){cout<<"n1";}
}

void myfun(){cout<<"outside";}

//using namespace n1; //error

int main(){
  myfun(); //outputs "outside"
  n1::myfun(); //outputs "n1"
}
```

24 Standard Template Library

The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.