

Informatics

GianAndrea Müller

June 7, 2018

CONTENTS

1	How to...	3	3.2.2	Smallest normalized and largest positive number	7	8.4	Character Literals	14	12.5	Recursion	21
1.1	... use this summary	3	3.2.3	Decimal to binary representation	7	8.5	String Literals	14	12.5.1	Call stack	21
1.2	... correct compilation errors	3	3.2.4	Binary to decimal representation	8	9	Operators	14	13	Arrays	22
1.3	... correct runtime errors	3	3.2.5	Guidelines when using floating point arithmetics	8	9.1	Arithmetic Operators	14	13.1	Arrays and Pointers	22
1.4	... approach problems	3	3.3	Hexadecimal numbers	8	9.1.1	Div-Mod identity	14	13.2	Multidimensional Arrays	23
1.5	... find more information	3	3.3.1	Hex nibbles	8	9.1.2	Pre- and post-in-/decrement	14	13.3	Passing arrays as arguments	23
2	Nice to know	4	4	Extended Backus-Naur-Form	8	9.2	Relational Operators	15	14	vector	23
2.1	Preprocessor directives	4	5	Character encoding	9	9.3	Logical Operators	15	14.1	Multidimensional vectors	23
2.1.1	Macro Definitions	4	5.1	ASCII	9	9.3.1	De Morgan's laws	15	14.2	Access	24
2.1.2	Conditional inclusions	4	6	Syntax	9	9.3.2	Application: XOR	15	14.3	Iterators	24
2.1.3	Line control	5	6.1	Basic program	9	9.3.3	Short circuit evaluation	15	14.4	Passing vectors to functions	24
2.1.4	Error directive	5	6.2	Identifiers	9	9.4	Bitwise Operators	15	15	Strings	24
2.1.5	Source file inclusion	5	6.3	Comments	9	9.5	Assignment Operators	16	15.1	C-style character string	24
2.1.6	Pragma directive	5	6.4	Data Types	10	9.6	Assignment Operators	16	15.2	string	25
2.1.7	Predefined macro names	5	6.4.1	Primitive Types	10	9.7	Misc Operators	16	16	Pointers	25
3	Positional Notation	6	6.4.2	Type modifiers	10	9.8	Operator Precedence and Associativity	16	16.1	Pointer arithmetic	25
3.1	Binary numbers	6	6.4.3	Find type sizes on your system	11	10	Conditional Statements	17	16.2	Const Pointers	25
3.1.1	From Decimal to Binary	6	6.4.4	Find minimum and maximum value of int	11	10.1	if	17	16.3	Additional Information	26
3.1.2	From Binary to Decimal	6	6.4.5	Type qualifiers: const volatile restrict	11	10.1.1	if...else	18	16.4	Differences between Pointers and References	26
3.1.3	Two's complement	6	6.4.6	Storage classes	11	10.1.2	if...else if... else	18	17	References	26
3.1.4	Overflow and Underflow	6	6.4.7	Variable Definition	12	10.2	switch	18	18	Input/Output	26
3.1.5	Estimation of the order of magnitude	7	6.4.8	Variable Declaration	12	10.3	? : Operator	18	18.1	cout and cin	26
3.1.6	Orders of magnitude of data	7	6.4.9	typedef and using	12	11	Loop Types	18	18.2	ifstream and ofstream	27
3.2	Floating point numbers	7	6.4.10	union	12	11.1	while	18	18.2.1	opening a file	27
3.2.1	Description	7	6.4.11	enum	12	11.2	for	19	18.2.2	Reading from a file	27
			7	Variable Scope	13	11.3	do...while	19	18.2.3	Closing a file	27
			7.1	Blocks	13	11.4	break continue	19	18.2.4	Writing to a file	27
			7.2	Local Variables	13	12	Functions	19	19	struct	28
			7.3	Global Variables	13	12.1	Structure	19	20	class	29
			8	Literals	14	12.1.1	void	19	20.1	Object oriented programming	29
			8.1	Integer Literals	14	12.2	Declaration and Definition	19	20.2	Basics	29
			8.2	Floating-point Literals	14	12.2.1	return statement	20	20.3	Class Members	29
			8.3	Boolean Literals	14	12.3	Calling a function	20	20.4	Class Access Modifiers	29
						12.4	Function Arguments	20	20.4.1	Access methods	29
						12.4.1	Call by Value	20	20.4.2	Access operators	30
						12.4.2	Call by Reference	20	20.4.3	This pointer	30
						12.4.3	Return by Reference	20	20.5	Constructors	30
						12.4.4	Default Values for Parameters	21	20.5.1	Copy Constructor	30

20.6	Destructor	31
20.7	friend	31
20.8	Static Members	31
21	Inheritance	31
21.1	Access Control and Inheritance	31
21.2	Polymorphism	31
22	Overloading	31
22.1	Function overloading	31
22.2	Operator overloading	32
22.2.1	Overloadable operators	32
22.2.2	Chained operators . . .	32
23	Dynamic Memory	33
24	Namespaces	33
25	Templates	33
25.1	Template function	33
25.2	Template class	33
26	Standard Template Library	34

1 HOW TO...

1.1 ... USE THIS SUMMARY

This summary is an overview of the functionality of C++ in connection with the informatics course for mechanical engineers. It covers the content of the lectures but also contains additional information.

To emphasize the connection to the lecture all chapters containing purely additional information are marked in blue.

1.2 ... CORRECT COMPILATION ERRORS

Read error messages, review basic syntax, look for the additional semicolon.

1.3 ... CORRECT RUNTIME ERRORS

Use a [debugger](#).

Or include safeguards in your code, checking the state of variables during runtime:

```
1 // #define NDEBUG // uncomment to ignore assertions
2 #include <cassert>
3
4 int main(){
5     cout<<"Enter 0 or 1"<<endl;
6     int a;
7     cin>>a;
8     assert( a == 1 || a == 0 );
9 }
```

1.4 ... APPROACH PROBLEMS

1. Define your problem.
2. Find your algorithm.
3. Code feature.
4. Compile.
5. goto 3.

1.5 ... FIND MORE INFORMATION

- [Comprehensive Tutorial](#)
- [User friendly documentation](#)
- [Extensive technical documentation](#)

TERMS

Algorithm An algorithm is a set of rules that defines a sequence of operations to get 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 document or fragment 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 cpp-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 and body.

Literals Literals represent constant value 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 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 and binary operators.

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

2 NICE TO KNOW

2.1 PREPROCESSOR DIRECTIVES

Preprocessor directives are lines preceded by a **#**. These lines are not program statements but directives for the preprocessor, thus are evaluated before the program is compiled. These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

2.1.1 MACRO DEFINITIONS

```
1 #define identifier replacement
```

When the preprocessor encounters this directive it replaces any occurrence of **identifier** in the rest of the code by **replacement**. A macro lasts until it is undefined with **#undef**. As seen in the following example it is also possible to define functions:

```
1 //A simple constant
2 #define TABLE_SIZE 100
3 int table1 [TABLE_SIZE];
4 #undef TABLE_SIZE //lasts until here
5
6 //A function
7 #define getmax(a,b) a>b?a:b
8
9 int main(){
10     int x = 5, y;
11     y = getmax(x,2); //replaced as: y = x>2?x:2
12 }
13
14 }
```

This would replace any occurrence of **getmax** followed by two arguments by the replacement expression, but also replace each identifier by its respective argument.

```
1 #define str(x) #x
2
3 cout<<str(test); // replaced as: cout<<"test";
```

As seen above an identifier preceded by **#** will be replaced by the argument in double quotes.

```
1 #define glue(a,b) a ## b
2 glue(c,out) << "test"; //replaced as: cout << "test";
```

The operator **##** concatenates two arguments leaving no white space between them.

2.1.2 CONDITIONAL INCLUSIONS

The directives **#ifdef**, **#ifndef**, **#if**, **#endif**, **#else** and **#elif** allow to include or discard part of the code if a certain condition is met.

#ifdef allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. **#endif** ends the conditional block.

```
1 #ifdef TABLE_SIZE
2 int table[TABLE_SIZE];
3 #endif
```

#ifndef serves for the exact opposite: the code between **#ifndef** and **#endif** directives is

only compiled if the specified identifier has not been defined yet.

```
1 #ifndef TABLE_SIZE
2 #define TABLE_SIZE 100
3 #endif
4 int table[TABLE_SIZE];
```

The `#if`, `#else` and `#elif` directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` and `#elif` can only evaluate constant expressions, including macro expressions.

```
1 #if TABLE_SIZE>200
2 #undef TABLE_SIZE
3 #define TABLE_SIZE 200
4
5 #elif TABLE_SIZE<50
6 #undef TABLE_SIZE
7 #define TABLE_SIZE 50
8
9 #else
10 #undef TABLE_SIZE
11 #define TABLE_SIZE 100
12 #endif
13
14 int table[TABLE_SIZE];
```

Notice that the compelling advantage of preprocessor directives over normal conditional statements is that preprocessor directives are evaluated before the code is compiled. An interesting application of that concept is the making of different versions of a program, for instance one version that has special debugging precautions and a second version which runs faster but omits these safety measures.

2.1.3 LINE CONTROL

When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place.

```
1 // #line number "filename"
2
3 #line 20 "assigning variable"
4 int a?;
```

This code will generate an error that will be shown as error in file "assigning variable", line 20.

2.1.4 ERROR DIRECTIVE

The `#error` directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter.

```
1 #ifndef _cplusplus
2 #error A C++ compiler is required!
3 #endif
```

2.1.5 SOURCE FILE INCLUSION

The `#include` directive is replaced by the entire content of the specified header or file. There are two ways to use `#include`:

```
1 #include <header>
2 #include "file"
```

In the first case, a header is specified between angle-brackets `<>`. This is used to include headers provided by the implementation, such as the headers that compose the standard library (`iostream`, `string`,...). Whether the headers are actually files or exist in some other form is implementation-defined, but in any case they shall be properly included with this directive.

The syntax used in the second `#include` uses quotes, and includes a file. The file is searched for in an implementation-defined manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes (`"`) were replaced by angle-brackets (`<>`).

2.1.6 PRAGMA DIRECTIVE

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`.

2.1.7 PREDEFINED MACRO NAMES

<code>__LINE__</code>	Integer value representing the current line in the source code file being compiled.
<code>__FILE__</code>	A string literal containing the presumed name of the source file being compiled.
<code>__DATE__</code>	A string literal in the form “Mmm dd yyyy” containing the date in which the compilation process began.
<code>__TIME__</code>	A string literal in the form “hh:mm:ss” containing the time at which the compilation process began.
<code>__cplusplus</code>	An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler.

3 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.

3.1 BINARY NUMBERS

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

3.1.1 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.

91310 = 10 * 9131 +	0	61 = 2 * 30 +	1
9131 = 10 * 913 +	1	30 = 2 * 15 +	0
913 = 10 * 91 +	3	15 = 2 * 7 +	1
91 = 10 * 9 +	1	7 = 2 * 3 +	1
9 = 10 * 0 +	9	3 = 2 * 1 +	1
		1 = 2 * 0 +	1

3.1.2 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.

							Σ
Ziffern	1	1	1	1	0	1	
Multiplikator	32	16	8	4	2	1	
Wert	32	16	8	4	0	1	61

3.1.3 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 $2^{N-1} - 1$.

3.1.4 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:

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

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.

3.1.5 ESTIMATION OF THE ORDER OF MAGNITUDE

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

3.1.6 ORDERS OF MAGNITUDE OF DATA

Multiple of bytes					
Decimal			Binary		
Value		Metric	Value		IEC
1000	kB	kilobyte	1024	KiB	kibibyte
1000 ²	MB	megabyte	1024 ²	MiB	mebibyte
1000 ³	GB	gigabyte	1024 ³	GiB	gibibyte
1000 ⁴	TB	terabyte	1024 ⁴	TiB	tebibyte
1000 ⁵	PB	petabyte	1024 ⁵	PiB	pebibyte
1000 ⁶	EB	exabyte	1024 ⁵	EiB	exbibyte

3.2 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 limit- edness. 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.

3.2.1 DESCRIPTION

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

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

Where the asterisk, when present, signalsises that the represented number always start 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. Of course this definition

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:

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

3.2.2 SMALLEST NORMALIZED AND LARGEST POSITIVE NUMBER

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

3.2.3 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
	\vdots	
1.11100		

3.2.4 BINARY TO DECIMAL REPRESENTATION

binary:	1	1	1	1	.	1	1	1
decimal:	8	4	2	1		$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$

3.2.5 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!

3.3 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.

3.3.1 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

Beispiel mit 32-bit Zahlen: 0x00000000 – 0xffffffff

- 0x80000000: höchstes Bit einer 32-bit Zahl gesetzt
- 0xffffffff: alle Bits einer 32-bit Zahl gesetzt

4 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 with a simple syntax what sentences can be built with the words of another language. 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:

```

1 digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6"
  | "7" | "8" | "9" ;
2 digit = "0" | digit excluding zero ;
3 natural number = digit excluding zero, {digit};
4 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.

5 CHARACTER ENCODING

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

5.1 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:

(65)–	10 00001 =	'A'
(66)–	10 00010 =	'B'
	⋮	
(97)–	11 00001 =	'a'
(98)–	11 00010 =	'b'

6 SYNTAX

6.1 BASIC PROGRAM

```
1 #include <iostream>
2 // #include "local_header_file.h"
3
4 /*
5  * Function declarations (and definitions)
6  */
7
8 int main(int argc, char ** argv)
9 {
10     /*
11      * Function calls
12      */
13     std::cout << "Hello World!" << std::endl;
14     return 0;
15 }
16
17 /*
18  * Function definitions
19  */
```

6.2 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.

More information on [Identifiers](#).

6.3 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:

```

1 // normal comment
2
3 /*
4 multi
5 line
6 comment
7 */

```

Both versions can be nested:

```

1 ///*comment in a comment*/
2
3 /*
4 cout<<"Hello World!<<endl; //comment in a comment
5 */

```

6.4 DATA TYPES

6.4.1 PRIMITIVE TYPES

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void

bool bool is used to save logical values. Check out section 9.3.

char char is used to represent characters and relies on encoding in ASCII. See section 5 for more detail.

int int is used to save integer numbers. For clarification check out section 3.

float & double float and double are used to save floating point numbers. For clarification check out section 3.2.

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.

```

1 void print(int a){
2     std::cout<<a<<std::endl;
3 }

```

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

```

1 void print(void){
2     std::cout<<"I'm pretty lame."<<std::endl;
3 }

```

3. **Declaration of universal pointers:** When used in the declaration of a pointer, void specifies that this pointer is universal, which means that it can point to any other type of data which is not declared with const or volatile. A void pointer cannot be dereferenced unless it is cast to another specific type.

```

1 void * universal_pointer;
2 int i = 3;
3 universal_pointer = &i;
4 //cast to pointer of type int
5 int * int_pointer = (int *)universal_pointer;

```

6.4.2 TYPE MODIFIERS

There exist a number of type modifiers:

Modifier	Effect
signed	variable interpreted as signed
unsigned	variable interpreted as unsigned
short	half number of allocated bits if possible
long	double number of allocated bits if possible

Based on the primitive types and their modifiers the spectrum of available types can be established. Their sizes differ depending on compiler and environment.

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)

6.4.3 FIND TYPE SIZES ON YOUR SYSTEM

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout<< "Size: char : "<<sizeof(char)<<endl;
6     cout<< "Size: int : "<<sizeof(int)<<endl;
7     cout<< "Size: short int : "<<sizeof(short int)<<endl;
8     cout<< "Size: long int : "<<sizeof(long int)<<endl;
9     cout<< "Size: float : "<<sizeof(float)<<endl;
10    cout<< "Size: double : "<<sizeof(double)<<endl;
11
12    return 0;
13 }
```

6.4.4 FIND MINIMUM AND MAXIMUM VALUE OF INT

```

1 #include <iostream>
2 #include <limits>
3 using namespace std;
4
5 int main() {
6     cout << "Minimum int value is "
7         << numeric_limits<int>::min() << ".\n"
8         << "Maximum int value is "
9         << numeric_limits<int>::max() << ".\n";
10    return 0;
11 }
```

6.4.5 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**!

volatile The modifier **volatile** tells the compile that a variable's value may be changed in ways not explicitly specified by the program. (Imagine for instance manually changing the position of a switch on a microprocessor.)

restrict A pointer qualified by **restrict** is initially the only means by which the object it points to can be accessed.

6.4.6 STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify.

auto The auto storage class is the default storage class for all local variables.

register The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

static The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

extern The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

mutable The mutable storage class only applies to non-static class members of non-reference and non-const type. When a class member is declared mutable it can be changed by const member functions.

6.4.7 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. Type has to be either a primitive type or a user defined object. Multiple names can be introduced at once if separated by commas.

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

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

```
1 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.

6.4.8 VARIABLE DECLARATION

It is possible to declare a variable without defining it. The declaration is accepted during compilation but has to be fitted with a definition at the time of linking of the program. This means that if a program consists of multiple files you can declare your variable wherever you need it but only define it once, since multiple definitions of the same variable are prohibited.

```
1 //Variable declaration
2 extern int a,b; //Compiler knows that the variable exists
3
4 //Variable definition
5 int a,b; //Compiler knows that the variable exists
6           //And allocates the storage space needed
```

6.4.9 TYPEDEF AND USING

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

```
1 //typedef type newname;
2
3 typedef unsigned int uint;
4
5 //using newname = type;
6
7 using uchar = unsigned char;
8
9 uint a = 3; //a is an unsigned int
10
11 uchar b = '2'; //b is an unsigned char
```

6.4.10 UNION

A union is a special class type that can hold only one of its non-static data members at a time. The union is only as big as necessary to hold its largest data member. The other data members are allocated in the same bytes as part of that largest member. The details of that allocation are implementation-defined, and it's undefined behavior to read from the member of the union that wasn't most recently written.

6.4.11 ENUM

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. This can improve readability of the code as values are given meaning.

```
1 //enum enum-name {list of names} var-list;
2
3 enum day {mon, tue, wed, thu, fri, sat, sun} today;
4 today = mon; //today == 0
5 today = sun; //today == 6
```

The values assigned to the names start with zero and increment by default. But you can also directly assign values:

```
1 enum color {red, green = 5, blue};
2 color c;
3 c = red;    //c == 0
4 c = green;  //c == 5
5 c = blue;   //c == 6
```

7 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)

7.1 BLOCKS

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

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

Control statements generate blocks as well:

```
1 for (int i = 0; i < 10; i++){
2     cout<<i<<endl;
3 }
4 // cout<<i<<endl; //not possible i is out of scope
```

7.2 LOCAL VARIABLES

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

```
1 for(int i = 0; i<3; i++){ //scope of the variable i
2     std::cout<<i<<" ";
3 } //end of scope
4
5 // std::cout<<i; will result in error
```

A variable can be redefined within a block. In that case the “closer” definition is used.

```
1 int i = 5;
2 {
3     int i = 3;
4     std :: cout << i; // outputs 3
5 }
6 std :: cout << i; // outputs 5
```

7.3 GLOBAL VARIABLES

Global variables are defined outside of all the functions, usually on top of the program. They will hold their value throughout the life-time of your program. This also means that all functions within your program can access these variables.

Including global variables the case of maximum complication, however undesired, is the following:

```
1 #include <iostream>
2 using namespace std;
3
4 int i = 2;
5
6 void fun(){
7     cout<<i;
8 }
9
10 int main(){
11     int i = 5;
12     {
13         int i = 3;
14         std :: cout << i;    // outputs 3
15     }
16     std :: cout << i;        // outputs 5
17     fun();                  // outputs 2
18 }
```

8 LITERALS

8.1 INTEGER LITERALS

An integer literal can be a binary, decimal, octal or hexadecimal constant. A prefix specifies the base: `0b` for binary, `0` 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).

```
1 212      // Decimal number
2 212u1    // Long unsigned decimal number
3 0xFFeL   // Long hexadecimal number
4 0b101    // Binary representation of 5
5 011      // Octal representation of 9
6 078      // Illegal: 8 is not an octal digit
```

8.2 FLOATING-POINT LITERALS

A floating point literal has an integer part, a decimal point, a fraction 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`.

```
1 3.14195  // Decimal representation of pi
2 314195E-5L // Exponential representation of pi
3 510E     // Illegal: incomplete exponent
4 210f     // Illegal: no decimal or exponent
5 .e55     // Illegal: missing integer or fraction
```

8.3 BOOLEAN LITERALS

There are two Boolean literals and they are part of standard C++ keywords: `true` and `false`.

```
1 bool a = true;
2 cout<<a; //outputs 1
3 a = false;
4 cout<<a; //outputs 0
5 bool = some_integer; //true for some_integer != 0
```

The conversion from integer to `bool` results in `true` for all nonzero values and in `false` for zero.

8.4 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'`).

<code>\\</code>	<code>\\</code> character	<code>\f</code>	Form feed
<code>\'</code>	<code>'</code> character	<code>\n</code>	Newline
<code>\"</code>	<code>"</code> character	<code>\r</code>	Carriage return
<code>\?</code>	<code>?</code> character	<code>\t</code>	Horizontal tab
<code>\a</code>	Alert or bell	<code>\v</code>	Vertical tab
<code>\b</code>	Backspace		

8.5 STRING LITERALS

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

9 OPERATORS

9.1 ARITHMETIC OPERATORS

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

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

9.1.1 DIV-MOD IDENTITY

$$a/b * b + a\%b == a$$

9.1.2 PRE- AND POST-IN-/DECREMENT

The operators used for increment and decrement each have two tasks. They 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:** Increase the value of the argument and return the previous value.

The signatures of those two functions are as follows:

```
1 //pre-increment outside class declaration
2 T & operator++(T& a);
3 //post-increment outside class declaration
4 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:

```
1 //pre-increment: return by reference
2 int& operator++(int& a){
3     a = a + 1;
4     return a;
5 }
6
7 //post-increment: return by value
8 int operator++(int& a, int i){
9     int temp = a;
10    a = a + 1;
11    return temp;
12 }
```

9.2 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.

9.3 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 reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make false.

9.3.1 DE MORGAN'S LAWS

$$\begin{aligned}!(a \ \&\& \ b) &== (!a \ || \ !b) \\!(a \ || \ b) &== (!a \ \&\& \ !b)\end{aligned}$$

9.3.2 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

9.3.3 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. **Thus the simpler evaluation should always be on the left of the operator.**

9.4 BITWISE OPERATORS

Bitwise operators work on bits and perform bit-by-bit operations. The truth tables for bitwise AND `&`, bitwise OR `|` and bitwise exclusive OR (XOR) `^` are:

p	q	p&q	p q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

<code>&</code>	Binary AND Operator copies a bit to the result if it exists in both operands.
<code> </code>	Binary OR Operator copies a bit if it exists in either operand.
<code>^</code>	Binary XOR Operator copies the bit if it is set in one operand but not both.
<code>~</code>	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<code><<</code>	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
<code>>></code>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

9.5 ASSIGNMENT OPERATORS

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

9.6 ASSIGNMENT OPERATORS

<code><<=</code>	Left shift AND assignment operator.
<code>>>=</code>	Right shift AND assignment operator.
<code>&=</code>	Bitwise AND assignment operator.
<code>^=</code>	Bitwise exclusive OR and assignment operator.
<code> =</code>	Bitwise inclusive OR and assignment operator.

9.7 MISC OPERATORS

<code>sizeof</code>	The sizeof operator returns the size of a variable in bytes. For example, <code>sizeof(a)</code> , where 'a' is integer, and will return 4.
<code>?X:Y</code>	If Condition is true then it returns value of X otherwise returns value of Y.
<code>,</code>	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.
<code>.</code>	The member operator <code>.</code> accesses a member of the operand on the left as specified by the operator on the right.
<code>-></code>	The member operator <code>-></code> 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 <code>int(2,200)</code> would return 2.
<code>&</code>	The address operator returns the address of a variable.
<code>*</code>	The dereference operator returns the value the operand points to.

9.8 OPERATOR PRECEDENCE AND ASSOCIATIVITY

P.	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	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[] delete delete[]	Dynamic memory allocation 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

```

1 cout<<a&&b;      //(cout<<a)&&b;
2
3 *p++             //(p++);
4
5 a = b = c = d;   //a = (b =(c = d));
6
7 a + b - c;       //(a + b) - c;
8
9 delete ++*p;     //delete(++(*p))

```

1. By its precedence << is evaluated before &&.
2. By its precedence ++ is evaluated before *.
3. Operators with the same precedence are evaluated based on their associativity. For right-to-left associative operators as =, the evaluation proceeds from right to left. Thus the assignments made in line 5 are in the order of their execution: c = d; which returns a reference to c, b = c; which returns a reference to b and a = b;.
4. Operators with the same precedence are evaluated based on their associativity. For left-to-right associative operators as + and - the evaluation proceeds from left to right.
5. ++(), *() and delete have the same precedence, and are thus evaluated based on their associativity, which is right-to-left. Therefore ++() is evaluated after *() and delete is evaluated last.

If written badly expression can result in undefined behaviour:

```

1 f(++i, ++i);
2 n = ++i + i;
3 b = ++a - a++;

```

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

10 CONDITIONAL STATEMENTS

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

10.1 IF

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

```

1 if (condition) single_statement;
2
3 if (condition) {
4     statement1;
5     statement2;
6 }

```

10.1.1 IF...ELSE

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

```

1 if (condition) {
2     statement1;
3 }
4 else {
5     statement2;
6 }

```

10.1.2 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.

```

1 if (condition1) {
2
3 }
4 else if (condition2) {
5
6 }
7 else {
8
9 }

```

10.2 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).

```

1 switch (expression) {
2     case 1 : cout << '1'; // prints "1"
3     case 2 : cout << '2'; // then prints "2"
4 }
5
6 switch (expression) {
7     case 1 : cout << '1'; // prints "1"
8         break; // then exits the switch
9     case 2 : cout << '2';
10        break;
11    default : cout << "default";
12 }

```

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.

10.3 ? : OPERATOR

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

```

1 a>b?a:b;
2
3 if(a>b) {
4     return a;
5 }
6 else {
7     return b;
8 }

```

The code snippets above accomplish exactly the same.

11 LOOP TYPES

11.1 WHILE

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

```

1 while (condition) {
2     statement;
3 }

```

11.2 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.

```
1 for (init-statement; condition; expression){
2     statement;
3 }
4
5 for (int i = 0; i < n ; i++){
6     statement;
7 }
8
9 for (;condition;) {
10    //equivalent to while(condition)
11 }
```

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

11.3 DO...WHILE

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

```
1 do {
2     statement
3 } while(condition);
```

11.4 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**).

```
1 while(true){
2     if(condition){
3         continue;
4     }
5     if(condition2){
6         break;
7     }
8 }
```

12 FUNCTIONS

12.1 STRUCTURE

```
1 return_type function_name (parameter list) {
2     body of the function
3 }
```

- **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.

12.1.1 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**;

12.2 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 follow by the definition.

```
1 return_type function_name ( parameter list );
2
3 //for example
4 int max (int, int);
```

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

```
1 int min (int a, int b){
2     return a<b?a:b;
3 }
```

Or come at some later point. Here we define the previously declared max function:

```
1 int max (int a, int b){
2     return a>b?a:b;
3 }
```

12.2.1 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 value given to the return statement has to match the return type defined in the declaration of the function.

12.3 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:

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

12.4 FUNCTION ARGUMENTS

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

12.4.1 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.

```
1 void change(int a){
2     a = 4;
3 }
```

12.4.2 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.

```
1 void change_ref(int & a){
2     a = 4;
3 }
```

When calling the function there is no visible difference to a call by value:

```
1 int main(){
2     int b = 3;
3     change(b); //no effect
4     change_ref(b); //b = 4;
5 }
```

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

```
1 void change_poi(int * a){
2     *a = 4;
3 }
```

```
1 int main(){
2     int b = 3;
3     change(b); // no effect
4     change_poi(&b); // b = 4;
5 }
```

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

```

1 int & increment (int & i){
2     i = i + 1;
3     //pass reference to variable that exists outside
4     return i;
5 }

```

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

```

1 int & increment (int no_reference){
2     no_reference = no_reference + 1;
3     //pass reference to local variable that
4     //will not persist beyond the function scope
5     return no_reference;
6 }

```

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 a variable that exists outside the function scope. It is possible to allocate new memory and return a pointer to that memory.

```

1 int * make_new_array (int Length){
2     int * array_pointer;
3     array_pointer = new int [Length];
4     return array_pointer;
5 }

```

12.4.4 DEFAULT VALUES FOR PARAMETERS

It is possible to define default values for function parameters. Those will be used when the corresponding argument is left blank when calling the function.

```

1 void count_to_ten_or_more (int n = 10){
2     for (int i = 0; i<n; i++){
3         cout<<i+1<<" ";
4     }
5     cout<<endl;
6 }

```

With the default value set, the function above can be called as `count_to_ten_or_more()`; which will result in a count to ten, or it can be given any argument of choice.

12.5 RECURSION

It is possible to define function that call themselves:

```

1 void f(){
2     f(); //function calls itself endlessly
3 }

```

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.

```

1 //POST: return value n!
2
3 unsigned int fac (unsigned int n)
4 {
5     //termination condition
6     if (n <= 1) return 1;
7
8     //recursion with direction n->1
9     return n * fac(n-1);
10 }

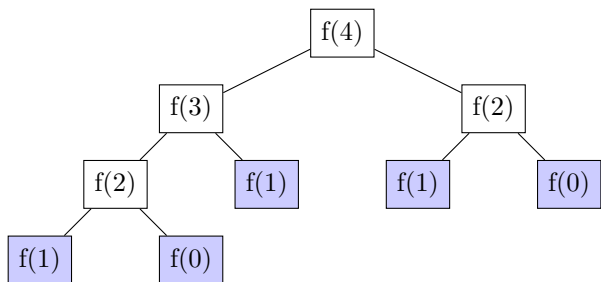
```

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

12.5.1 CALL STACK

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

```
1 //POST: return value is the n-th
2 //Fibonacci number F(n)
3 unsigned int fib(const unsigned int n){
4     if (n == 0) return 0;
5     if (n == 1) return 1;
6     return fib(n-1) + fib(n-2); //n>1
7 }
```



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

```
1 // POST: return value is the n-th Fibonacci number F(n)
2 unsigned int fib2 (const unsigned int n) {
3     if (n == 0) return 0;
4     if (n <= 2) return 1;
5     unsigned int a = 1; // F_1
6     unsigned int b = 1; // F_2
7     for (unsigned int i = 3; i <= n; ++i) {
8         unsigned int a_prev = a; // F_i-2
9         a = b; // F_i-1
10        b += a_prev; // F_i-1 += F_i-2 -> F_i
11    }
12    return b;
13 }
```

13 ARRAYS

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

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

13.1 ARRAYS AND POINTERS

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

```
1 int list_1 [4] = {1,2,3,4};
2 int list_2 [1];
3 // list_1 = list_2; // error
4 int * list_pointer_1 = list_1;
5 int * list_pointer_2 = &list_1[0];
6 list_1[1]; // value: 2
7 list_pointer_1[1]; //value: 2
8 list_pointer_2[1]; //value: 2
9 *(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:


```

1 int n;
2 cin >> n;
3 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!

13.2 MULTIDIMENSIONAL ARRAYS

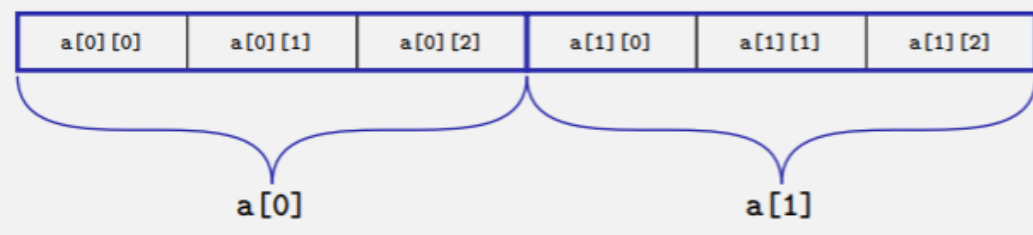
Arrays can have more than one dimension, and can thus also be used to represent matrices or tables.

```

1 //2 elements, each an array of length 3
2 //uninitialized
3 int a_1[2][3];
4 //initialized, size specified
5 int a[2][3] = {
6     {1,2,3},{2,6,3}
7 };
8 //initialized, size partially specified
9 int a[][3] = {
10     {1,3,4},{2,3,4},{13 3 2}
11 };

```

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:



13.3 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 simply 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:

```

1 void some_array_function(int pointer []){
2     //...
3 }
4
5 void nice_array_function(int (&arrayname) [10]){
6     //...
7 }

```

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.

14 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.

```

1 #include <vector>
2
3 std::vector<int> int_vector;

```

The basic functionality of vector can be seen here, [read](#) for more details.

```

1 //create a bool vector with length 100
2 //all entries set to false
3 std::vector<bool> status(100,false);
4
5 //create a copy of an existing vector
6 std::vector<bool> copy(status);
7
8 //access entries
9 copy[0]; // first entry
10 copy.at(0); // first entry
11 copy.push_back(true); // append entry: true
12 copy.pop_back(); //delete last element

```

14.1 MULTIDIMENSIONAL VECTORS

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

```
1 using namespace std;
2 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.

14.2 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.

```
1 //long array (syntax simplified!)
2 int a [] = {1,2,3,4,5,6,...,100};
3
4 //random access:
5 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 and forth in steps of one. Sequential access can be implemented as follows:

```
1 int a[5] = {1,2,3,4,5};
2 for (int* p = a; p < a+5; ++p)
3     cout<<*p<<" ";
4
5 //For vectors iterators can be used
```

14.3 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.

```
1 std::vector<int>::const_iterator
2
3 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.

```
1 std::vector<int> vec;
2 vec.push_back(1);
3 vec.push_back(2);
4 std::vector<int>::iterator it = vec.begin();
5
6 while(it<vec.end()) {
7     cout<<*it<<endl;
8     it++;
9 }
```

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.

14.4 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.

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

15 STRINGS

15.1 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.

```

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

```

15.2 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.

```

1 #include <string>
2 std::string text1 = "bool";
3 text1.length(); //A string knows its length
4 //initialize with variable length n
5 //and fill with 'a'
6 std::string text2 (n, 'a');
7 //string understands comparisons
8 //and many other operations
9 text1 == text2;

```

16 POINTERS

Pointers are variables saving addresses of other variables.

```

1 int a = 6;
2 //b is a pointer to an int
3 //and takes the address of a
4 int * b = &a;
5
6 (*b)++; //a == 7

```

There are three important operators for pointers:

- Declaration of a pointer:

```

1 //<type> * <name>;
2 int * pointer;

```

- Address of a variable:

```

1 //& <variable>;
2
3 int var;
4 pointer = & var;

```

- Accessing the value pointed to (dereferencing)

```

1 //want to set var = 7;
2 //*<initialised_pointer>;
3 *pointer = 7; //var == 7

```

16.1 POINTER ARITHMETIC

In connection with arrays there are more operations possible:

```

1 int my_array [10];
2 //Pointer to first element
3 int my_pointer = my_array;
4
5 while(my_pointer<my_array+10){
6     //do something
7     //increment pointer
8     my_pointer++;
9 }

```

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.

16.2 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.

16.3 ADDITIONAL INFORMATION

Also see section 13.1 for the connection between pointers and arrays as well as 12.4.2 to see how pointers can be used to implement call by reference.

Also consider the additional [information](#) explaining many advanced uses of pointers.

16.4 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.

17 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!**

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

In c++ a reference declaration consists of the type which is to be 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`!

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

Only const references are allowed to point to R-values!

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

Only const references can refer to constant variables.

18 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.

18.1 COUT AND CIN

```
1 #include <iostream>
2
3 //allows omitting std:: before cout and cin
4 using namespace std;
5
6 int main(){
7     //declare variable
8     int a;
9     //initialise via keyboard input
10    cin>>a;
11    //print via console
12    cout<<a;
13 }
```

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:

```

1 //Stream operators
2 //PRE: Takes a ostream as an argument
3 //POST: Prints to the handed ostream
4 std::ostream& operator<<(std::ostream& os, const Rational
   & rational)
5 {
6     os << rational.numerator << "/" << rational.
       denominator;
7     return os;
8 }

```

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!

18.2 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.

18.2.1 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.

```

1 #include <iostream>
2 #include <fstream>
3
4 using namespace std;
5
6 int main(){
7     //declare an instance of ifstream
8     ifstream infile;
9     //open a file
10    infile.open("file_name.txt", ios::in);

```

18.2.2 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:

```

1 char in;
2 string whole_file;
3 //extract from file until
4 //end of file is reached
5 while(!infile.eof()){
6     //read next char
7     infile>>in;
8     //save char in string
9     whole_file.push_back(in);
10 }

```

18.2.3 CLOSING A FILE

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

```

1 infile.close();

```

18.2.4 WRITING TO A FILE

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

```

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

```

The we use outfile to write to a new file.

```

1 //After opening writing is possible
2 while(!whole_file.empty()){
3     char temp = whole_file.pop_back();
4     //Some kinda caesar code
5     temp++;
6     //Write to file
7     outfile<<temp;
8 }

```

After that the file is closed.

```

1 outfile.close();
2 return 0;
3 }

```

19 STRUCT

Struct allows the definition of new custom data types.

```

1 /*
2 struct <name> {
3     <type> <member_name>;
4     .
5     .
6     .
7 }<optional_instance_name>;
8
9 */
10
11 struct rational{
12     int n;
13     int d;
14 }a;
15
16 rational b;

```

- Now the new type `rational` is defined.
- It can be directly instantiated by inserting one or several 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.
- Member types can be combined in any way desired.

Members can be initialized directly:

```

1 //uninitialized
2 rational s;
3
4 //member-by-member
5 rational t = {1,5};
6
7 //copy member-by-member
8 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 22.2.

20 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 are private by default.

20.1 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.

20.2 BASICS

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

20.3 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.

```
1  /*
2  class <class_name> {
3      //member variable
4      <type> <member_name>;
5      //member function
6      <rtype> <member_fun_name> ();
7  }<optional_instance_name>;
8  */
9
10 class Vector {
11     //member definition
12 }a;
```

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

```
1  int main(){
2      Vector b;
3  }
```

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

20.4 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!

```
1  class Vector {
2      //default: private
3      double x; //is private
4  private:
5      double y; //is private
6  public:
7      void print(); //is public
8  };
```

20.4.1 ACCESS METHODS

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

```
1      //within definition of vector
2      double get_x() const {return x;}
3      double get_y() const {return y;}
4
5      void set_x(const double _x) {x = _x;}
6      void set_y(const double _y) {y = _y;}
```


20.4.2 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:

```
1 //Instance of Vector
2 Vector a;
3 //Pointer of type Vector to a
4 Vector * pointer = &a;
5
6 //Direct access operator .
7 a.set_x(7);
8
9 //Indirect access operator ->
10 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.

20.4.3 THIS POINTER

Sometimes it is useful to have access on the instance of a class as an object, not only on 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:

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

20.5 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.

```
1 class Vector {
2     double x,y;
3 public:
4     //Overwriting the default constructor
5     Vector () {
6         x = 0;
7         y = 0;
8     }
9     //Overloading the constructor
10    Vector (double _x, double _y)
11        : x(_x),y(_y) {}
12 }
```

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

20.5.1 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.
- copy 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.

```
1 /*
2 <classname> (const <classname> & obj) {
3     //body of constructor
4 }
5 */
6
7 Vector (const Vector& obj){
8     this->x = obj.x;
9     this->y = obj.y;
10 }
```

20.6 DESTRUCTOR

- Standard member function, exists in default form if not overwritten.
- Used to deallocate memory taken by an instance upon its deletion.
- 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.

```
1  /*
2  ~<classname> () {
3      delete <dynamic_variable>;
4      delete [] <dynamic_array>;
5  }
6  */
7
8  ~Vector () {
9      cout<<"Vector is being destructed."<<endl;
10 }
```

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.

20.7 FRIEND

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

```
1  //outside class
2  void foreign_print(Vector obj){
3      cout<<obj.x<<" "<<obj.y;
4  }
5
6  //inside class
7
8  friend void foreign_print(Vector obj);
```

20.8 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.

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

21 INHERITANCE

21.1 ACCESS CONTROL AND INHERITANCE

21.2 POLYMORPHISM

22 OVERLOADING

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

22.1 FUNCTION OVERLOADING

For 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.

```
1  //a way of identifying unknown variable types:
2
3  void identify (int a){
4      cout<<"My type is int."<<endl;
5  }
6
7  void identify (double a){
8      cout<<"My type is double."<<endl;
9  }
```

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

```
1  void fun();           //without argument
2  void fun(int a);      //with argument
3  void fun(double b);   //different type
4  //different number of arguments
5  void fun(int a, int b);
6  //different type of arguments
7  void fun(int a, double b);
8  //different order of arguments
9  void fun(double b, int a);
```

Based on the above the following overloads are not valid:

```

1 //No distinction by return type
2 int fun();
3 //No distinction by argument name
4 int fun(int different);

```

22.2 OPERATOR OVERLOADING

Operators 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`.

```

1 struct rational{
2     int n;
3     int d;
4 };

```

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

```

1 rational operator- (const rational& a, const rational& b)
2 {
3     rational result;
4     result.n = a.n*b.d - a.d*b.n;
5     result.d = a.d*b.d;
6     return result;
7 }

```

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

```

1 rational operator- (const rational& a){
2     rational result = a;
3     result.n = -result.n;
4     return result;
5 }

```

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:

```

1 int main(){
2     rational a = {1,3};
3     rational b = {2,4};
4     //unary operator-
5     rational c = -b; // -2/4
6     //binary operator-
7     rational d = a-b;
8 }

```

22.2.1 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](#)

22.2.2 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:

```

1 #include <iostream>
2
3 using namespace std;
4
5 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.

```

1 std::ostream& operator<< (std::ostream& out, rational r){
2     return out << r.n << "/" << r.d;
3 }

```

23 DYNAMIC MEMORY

24 NAMESPACES

25 TEMPLATES

Templates are the foundation of generic programming, which in different terms is the practice of writing code in a way that is independent of any particular data type. A template can be understood as a blueprint or a formula for creating a generic class or function. The first example of templates we got to enjoy was `vector` which was always handed its specific data type via angled brackets.

25.1 TEMPLATE FUNCTION

```

1 /* general form of a template
2 template <class type> ret-type func-name(parameter list)
3 {
4     // body of function
5 }
6 */
7 template <typename T>
8 T const& Max(T const& a, T const& b){
9     return a<b ? b:a;
10 }

```

In the example above we defined a function template `Max` that can now be used for any data type that has suitable overloading for the used operators. Without templates we would need to overload `Max` for every new datatype we'd like to feed it. When called a template function can recognize the data type it is fed and in contrast to template classes it does not need to be told explicitly what data type to expect as you can see below:

```

1 int main(){
2     int j = 29;
3     int k = 20;
4     Max(i,j); //returns 29
5
6     string s1 = "Hello";
7     string s2 = "World";
8     Max(s1,s2); //return "World";

```

25.2 TEMPLATE CLASS

```

1 /* general form a of a template class
2 template <class type> class class-name{
3     //class contents
4 }
5 */
6
7 //in the header file
8 template <class T>
9 class Stack {
10 private:
11     vector<T> elements;
12 public:
13     //push element
14     void push(T const&);
15     //pop element
16     void pop();
17     //return top element
18     T top() const;
19     //return true if empty
20     bool empty() const;
21 }

```

Above we have declared a template class that can handle stacks of any variable type we choose. Now we need to see how member functions are defined in this case:

```

1 //in the cpp file
2 template <class T>
3 void Stack<T>::push (T const& element){
4     // append copy of passed element
5     elements.push_back(element);
6 }

```

After having defined all member functions we can use our stack as follows:

```

1 Stack<int> intStack;
2
3 //manipulate our intStack
4 intStack.push(7);
5 cout << intStack.top() << endl;
6
7 Stack<string> stringStack;
8
9 stringStack.push("hello");
10 stringStack.push("world");
11 cout<<stringStack.top();
12 stringStack.pop();
13 cout<<stringStack.top();
14 //we just made ourselves
15 //a nice implementation of hello world ;)

```

26 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.