Informatics

GianAndrea Müller

June 5, 2018

Contents

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:

```
//#define NDEBUG  //uncomment to ignore assertions
#include <cassert>

int main(){
  cout<<"Enter 0 or 1"<<endl;
  int a;
  cin>>a;
  assert( a == 1 || a == 0 );
}
```

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

```
#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:

```
//A simple constant
#define TABLE_SIZE 100
int table1 [TABLE_SIZE];
#undef TABLE_SIZE //lasts until here

//A function
#define getmax(a,b) a>b?a:b

int main(){
  int x = 5, y;
  y = getmax(x,2); //replaced as: y = x>2?x:2
}

}
```

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

```
#define str(x) #x

cout<<str(test); // replaced as: cout<<"test";
```

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

```
#define glue(a,b) a ## b
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.

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#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.

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
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.

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

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.

```
//#line number "filename"

#line 20 "assigning variable"
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.

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

2.1.5 Source file inclusion

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

```
#include <header>
#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

`__LINE__`	Integer value representing the current line in the source code file being compiled.
`__FILE__`	A string literal containing the presumed name of the source file being compiled.
`__DATE__`	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
`__TIME__`	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
`__cplusplus`	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.

$$
\begin{aligned}
91310 &= 10 * 9131 + & 0 \\
9131 &= 10 * 913 + & 1 \\
913 &= 10 * 91 + & 3 \\
91 &= 10 * 9 + & 1 \\
9 &= 10 * 0 + & 9
\end{aligned}
\qquad
\begin{aligned}
61 &= 2 * 30 + & 1 \\
30 &= 2 * 15 + & 0 \\
15 &= 2 * 7 + & 1 \\
7 &= 2 * 3 + & 1 \\
3 &= 2 * 1 + & 1 \\
1 &= 2 * 0 + & 1
\end{aligned}
$$

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.

							$\sum$
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:

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

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.

3.1.5 Estimation of the order of magnitude

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

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 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.

3.2.1 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 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 Smalles 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
	⋮	

$1.1\overline{1100}$

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:

```
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.

5 Syntax

5.1 Basic program

```
#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
 */
```

5.2 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.

5.3 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:

```
// normal comment

/*
multi
line
comment
*/
```

Both versions can be nested:

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

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

5.4 Data Types

5.4.1 Primitive Types

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

char

5.4.2 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)

5.4.3 Find type sizes on your system

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

int main() {
    cout<< "Size: char : "<<sizeof(char)<<endl;
    cout<< "Size: int : "<<sizeof(int)<<endl;
    cout<< "Size: short int : "<<sizeof(short int)<<endl;
    cout<< "Size: long int : "<<sizeof(long int)<<endl;
    cout<< "Size: float : "<<sizeof(float)<<endl;
    cout<< "Size: double : "<<sizeof(double)<<endl;

    return 0;
}
```

5.4.4 Find minimum and maximum value of int

```cpp
#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;
}
```

5.4.5 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.

5.4.6 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.

5.4.7 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.

```
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.

5.4.8 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.

```
//Variable declaration
extern int a,b; //Compiler knows that the variable exists

//Variable definition
int a,b; //Compiler knows that the variable exists
         //And allocates the storage space needed
```

5.4.9 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
```

5.4.10 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.

5.4.11 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.

```
//enum enum-name {list of names} var-list;

enum day {mon, tue, wed, thu, fri, sat, sun} today;
today = mon; //today == 0
today = sun; //today == 6
```

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

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

6 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)

6.1 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
```

6.2 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
```

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

```
int i = 5;
{
  int i = 3;
  std :: cout << i; // outputs 3
}
std :: cout << i; // outputs 5
```

6.3 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 mean that all functions within your program can access these variables.
Including global variables the case of maximum complication, however undesired, is the following:

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

int i = 2;

void fun(){
  cout<<i;
}

int main(){
  int i = 5;
  {
    int i = 3;
    std :: cout << i;    // outputs 3
  }
  std :: cout << i;      // outputs 5
  fun();                 // outputs 2
}
```

7 Literals

7.1 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).

```
212      // Decimal number
212ul    // Long unsigned decimal number
0xFeeL   // Long hexadecimal number
0b101    // Binary representation of 5
011      // Octal representation of 9
078      // Illegal: 8 is not an octal digit
```

7.2 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.

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

7.3 Boolean Literals

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

```
bool a = true;
cout<<a; //outputs 1
a = false;
cout<<a; //outputs 0
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.

7.4 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		

7.5 String Literals

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

8 Operators

8.1 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 result of the division.

8.1.1 Div-Mod identity

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

8.1.2 Pre- and post-in-/decrement

8.2 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.

8.3 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. If a condition is true, then Logical NOT operator will make false.

8.3.1 De Morgan's laws

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

8.3.2 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

8.3.3 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.**

8.4 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

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

8.5 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.

8.6 Assignment Operators

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

8.7 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.
`,`	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.

8.8 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[]`	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

8.8.1 How to use this table

```
cout<<a&&b;     //(cout<<a)&&b;

*p++            //*(p++);

a = b = c = d; //a = (b =(c = d)));

a + b - c;      //(a + b) - c;

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:

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

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

9 Conditional Statements

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

9.1 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;
}
```

9.1.1 if...else

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

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

9.1.2 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 {

}
```

9.2 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.

9.3 ? : 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.

10 Loop Types

10.1 while

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

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

10.2 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.

10.3 do...while

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

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

10.4 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;
  }
}
```

11 Functions

11.1 Structure

```
return_type function_name (parameter list) {
  body 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.

11.1.1 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;`

11.2 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.

```
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 is followed by 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 come at some later point. Here we define the previously declared max function:

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

11.2.1 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.

11.3 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.

11.4 Function Arguments

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

11.4.1 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;
}
```

11.4.2 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;
}
```

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

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

11.4.3 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.**

```
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.

11.4.4 Call by Pointer

A call by pointer makes the parameter a copy of the address of the argument. Equivalent to a call by reference the changes on the argument persist. Contrasting to it, the parameter has to be dereferenced since it is a pointer.

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

When calling the function there is a difference to a call by value, the function must be handed an address!

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

11.4.5 Return by Pointer

In contrast to a return by reference, a return by pointer is 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.

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

11.4.6 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.

```
void count_to_ten_or_more (int n = 10){
  for (int i = 0; i<n; i++){
    cout<<i+1<<" ";
  }
  cout<<endl;
}
```

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.

11.5 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.

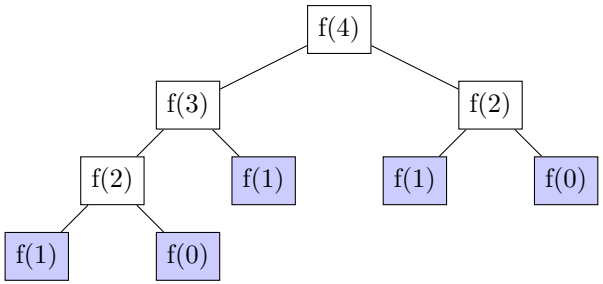
```
//POST: return value n!

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

11.5.1 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;
}
```

12 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.

12.1 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];
```

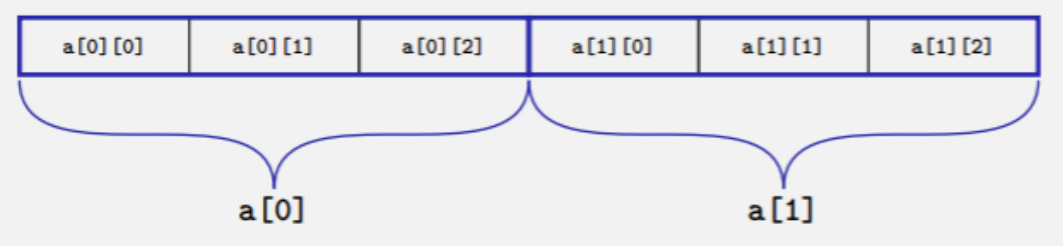
- The dynamically allocated array can now be used exactly the same as a static array.
- The dynamic array needs to be deleted with `delete [] dynamic_array` in order not to create memory leaks!

12.2 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:



12.3 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.

13 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
```

13.1 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.

13.2 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
```

13.3 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.

13.4 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);
```

14 Strings

14.1 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'};
```

14.2 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;
```

15 Pointers

16 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`!

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

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

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

Only `const` references can refer to constant variables.

17 Input/Output

18 struct

19 class

19.1 Class Members
19.2 Class Access Modifiers
19.3 Constructor and Destructor
19.4 Copy Constructor
19.5 friend
19.6 this
19.7 Static Members

20 Inheritance

20.1 Access Control and Inheritance

21 Overloading

21.1 Function overloading
21.2 Operator overloading
21.2.1 Overloadable operators

22 Polymorphism

23 Dynamic Memory

24 Namespaces

25 Templates

26 Signal Handling

27 Standard Template Library

28 Libraries

28.1 iostream
28.2 math
28.3 ctime