Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mechanics & Materials Laboratory
www.mm.ethz.ch

Computational Solid Mechanics

A Brief Introduction to C++

Last update: September 25, 2017

Instructor
Prof. Dr. Dennis M. Kochmann
Mechanics & Materials
ETH Zürich
Leonhardstr. 21, LEE N201
8092 Zurich
Switzerland
dmk@ethz.ch

Assistant
Raphaël Glaesener
Mechanics & Materials
ETH Zürich
Leonhardstr. 21, LEE N203
8092 Zurich
Switzerland
raphaegl@student.ethz.ch

Assistant
Abbas Tutcuoglu
Mechanics & Materials
ETH Zurich
Leonhardstr. 21, LEE N203
8092 Zurich
Switzerland
tabbas@ethz.ch

1 Disclaimer & Acknowledgements

The purpose of this script is to revive the fundamental concepts of the coding language C++. It can be seen as a short summary of the teaching material associated to *Informatik I* in the first bachelor semester of *Mechanical Engineering* at ETH. To gain a more advanced and deeper knowledge we strongly recommend *Einstieg in C++* written by Arnold Willemer for our German-speaking fellows, while *The C++ Programming Language* written by Bjarne Stroustrup is a good read with specific examples on the usage of the 2011 C++ standard (which soon might be outdated too though, given that the 2017 C++ standard is about to be released).

Furthermore, we would like to thank Greg Phlipot (*Caltech*) and Vidyasagar (*Caltech*) for kindly providing us with the code tutorials from previous lectures. Some of the coding examples in this script were inherited from their documentations.

We would also like to emphasize that a number of explanations and figures in this brief introduction to C++ have been copied or inspired by the lecture slides of M. Gross from the *Computer Science Department* at ETH for the lecture *Informatik I* at D-MAVT.

All the best,
Your TAs Raphaël Glaesener and Abbas Tutcuoglu

Five useful things to know at one glance

- C++ is a **case sensitive** language (variable1 $\neq$ Variable1 $\neq$ vArIaBlE1)
- Remember the **semicolon** at the end of the line
- C++ is **zero indexed**, meaning containers start at index 0
- **Compile time** - the time during which the source code is compiled into binary vs **Runtime** - the time during which the executable is run
- Two good **online resources for C++** are `www.cplusplus.com`, which provides structured introductions into numerous fields of C++ and `www.stackoverflow.com` - in case you have an issue, chances are someone else had it before you and found help from the community on stack overflow.

Suggested Coding Practice - all good things go by three

- Write **self-documenting** code, using detailed, **self-explanatory variable names**
- Have variable names ...
  - generally start with lower case letters
  - unless they're static in which case it is custom to start with a capital letter
  - or unless they are non-static public or private class members, in which case it is custom to prepend them by an underscore
- Indent your code **2 spaces "per level"** (e.g. the inside of a function, conditional-statement, loop, etc) - most development environments permit to define a tab as a number of spaces, which in this case should please be set to two to ensure consistency

Contents

2 Some important Unix commands

Command	**Meaning**
man *command*	Provides an instruction for different *command* and possible flags
info *command*	Provides a rather brief information about *command*
which *command*	Shows the absolute path to *command*
who	Shows all users that are currently signed in
ls	Shows all subdirectories and folders
ls -a	Additionally shows information about these subdirectories
cd *Path*	Used to navigate between the directories
cd..	Go back to superior directory
rm	Deletes files irrevocable
top	Shows all currently running processes
make *target*	follows the instructions in Makefile on what to do with *target* (in general to compile it)
make all	equivalent to performing 'make *target*' for all targets defined in Makefile
exit	sign out

Note: *command* in the above enumeration is a placeholder for any supported command
Note: option parameters as *-a* or *-anysupportedlettercombination* are oftentimes referred to as flags

3 Variable Types - Initialization & Declaration

3.1 Types

Complete list of fundamental types in C++		
Group	**Type names**	**Notes**
Character types	char	Exactly one byte in size. At least 8 bites.
Integer types (signed)	signed int	Not smaller than short. At least 16 bites
	signed long int	Not smaller than int. At least 32 bites
	signed long long int	Not smaller than long. At least 64 bites
Integer types (unsigned)	unsigned int	(same size as their signed counterparts)
	unsigned long int	
	unsigned long long int	
Floating-point types	float	Saves real numbers
	double	Precision not less than float
Boolean type	bool	Stores the information 'true' and 'false'
Void type	void	no storage
Auto type	auto	Compiler deduces type from initializer

Table 1: `http://www.cplusplus.com/doc/tutorial/variables/`

3.2 Initialization and declaration

Integers

```
int a;                  int a,b,c;              int a = 0;
int a = 0; b=1;         int a = b = c = 0       int a(0), b(1);
```

Chars

```
char 'Z';               char a = 90; (Ascii)
```

Double

```
double 7.34;            double 1/5.0;           double a = double (1/3);
```

3.3 Examples of valid and invalid declarations

Variable names may only contain:

- Alphabetic characters
- Numeric digits
- Underscores (_)

```
int firstname;              valid
int Firstname;              valid, variable differs from firstname
int leet1337;               valid
int 1337leet;               invalid, begins with a digit
int what_a_great_name;      valid
int var!;                   invalid, contains an illegal character
int _thisvar;               valid
```

4 Hello World

The following code structure compiles and executes. It is one of the simplest form of code:

```
1 // -*- C++ -*-
2
3 #include "/CSM/Definitions.h"
4   int main(int arc, char *argv[])
5   {
6    printf("Hello world!\n");
7    return 0;
8   }
```

5 Operators

There are 5 types of arithmetic operators: plus (+), minus (-), times (*), divide (/), Modulo (%)

5.1 Shortcut - operators

Addition and Subtraction Caution - in general, it may matter as to where the $--/++$ is placed:

```
a++;      ⇔    a = a+1;        a-;      ⇔    a=a-1;
a+=b;     ⇔    a = a+b;        a-=b;    ⇔    a=a-b;
```

Pre-increment: `++i` ⇒ First increases the value by one, then executes whichever series of commands it is subjected to.
Post-increment `i++` ⇒ First executes whichever series of commands it is subjected to, and only then increases the value by one.

```
#include<stdio.h>
#include<iostream>

using namespace std;

int main(){

  int i = 0;
  int a = (i++)*3;  // First i*3 is evaluated and then i is counted up
  int b = (++i)*2;  // First i is counted up and then i*2 is evaluated

  cout << "a should be zero - lets check: " << a << endl;
  cout << "b should be four - lets check: " << b << endl;

  return 0;
}
```

Multiplication and Division

```
a *= b ;          a = a*b;          a% = b ;
a = a%b;          a /= b ;          a = a/b;
```

Caution A: Division between two integers yields an integer - this is a very classy mistake

```
  int    a = 4;
  int    b = 3;
  int    c = a/b;
  double d = a/b;
  double e = (double)a /b;

  cout << "c should be 1       - lets check: " << c << endl;
  cout << "d should be 1       - lets check: " << d << endl;
  cout << "e should be 1.33333 - lets check: " << e << endl;
```

Caution B: Mod on neg. integers does not follow the standard definition from group theory.

```
  int a = -1; int amod3 = a%3;
  int b = +2; int bmod3 = b%3;

  cout << "a%3 = +2, but is -1 as we can verify here:" << amod3 << endl;
  cout << "b%3 = +2,           as we can verify here:" << bmod3 << endl;
```

5.2 Operators for booleans & Logical operators

Relational Expressions

In many cases one has to take advantage of comparative operations which are necessary for the decision making inside a code. C++ provides six relational operators to compare numbers. The result is either true (1) or false (0). Relational operators are evaluated after other arithmetic operations.

Operator	*Meaning*
`<`	is less than
`<=`	is less than or equal to
`==`	is equal to
`>`	is greater than
`>=`	is greater than or equal to
`!=`	is not equal to

Caution: Try not to compare doubles, floats, etc. using `==`. This absolutely makes sense for integers, but less so for floating point variables. Instead, take the absolute value of the difference and check if it is inferior to some tolerance bound.

6 Text-Output

Regardless of whether we simply want to greet the user, prompt him/her for input, return data or simply debug, text output is an invaluable tool. In favor of conserving the conciseness of this script, we limit ourselves to what we deem the three most important commands:

- **cout**: This is arguably the easiest-to-use output-on-the-console tool and is part of the *iostream*-library. Let's look at an example of how to use it:

```
#include<iostream>

using namespace std;

int main(){
  int b = 10;
  cout << "Return b: " << b << ". Line change:" << endl << "Once more..." << endl;
}
```

  Note, that 'endl' is representative of a line change. Further note, that in general we would have to write std::cout and std::endl, which we circumvent by 'using namespace std'.

  In the following we list some more formatting examples for outputting results:

Source code	*Output*
`cout « 3.14 « endl;`	3.14
`cout « 3.14159 « endl;`	3.14159
`cout « setw(10) « 3.14 « endl;`	3.14
`cout « setw(10) « 3.14159 « endl;`	3.14159
`cout.fill('*');`	
`cout « setw(10) « 3.14 « endl;`	******3.14
`cout.width(10);`	
`cout « "Hello" « endl;`	*****Hello

- **printf**: Only marginally more complex to use, but in our view significantly more versatile is `printf`, which comes hand in hand with the *stdio*-library. Let's take a look at a concrete example:

```
#include<stdio.h>

int main(){
  int b = 10;
  printf("Return b: %d. Line change:\nPerfetto! Once more...\n",b);
  printf("The major advantage of printf lies in its formatting capabilities.\n");
  printf("This will reserve three spaces for b: %3d.\n",b);
  printf("The same can be done for doubles: %4.2f %6.3f, %.2e",0.3,0.3,0.3);
}
```

  We immediately recognize that - much different to the previously presented `cout`, where we had to pause the strings, when inserting b or when changing lines - printf allows to finish the string and simply use placeholders to reserve space for what is defined after the string. Even more useful than that is the functionality of predefining the number of spaces that the output variables, i.e. including some sort of user-defined formatting. A nice summary for the different placeholders can be found in `http://www.cplusplus.com/reference/cstdio/printf/`.

- **fprintf:** Defined in the same library, `fprintf` supports the same capabilities as `printf` with the sole difference of providing the output in files and not on the console. The following example code gives a quick introduction of how a file can be openend and then be written upon using `fprintf`.

```c
#include<stdio.h>

int main(){
    FILE *myfirstfile;

    myfirstfile = fopen("MyFirstFile.csv","w");
    fprintf(myfirstfile,"time, pos, posX, posY, posZ, state\n");
    fprintf(myfirstfile,"%4.2f, %3d, %3d, %3d, %3d, %1d\n",0.01,17,1,0,1,2);
    fclose(myfirstfile);

    printf("Hupps. Completely forgot to add the last set of data!\n");
    myfirstfile = fopen("MyFirstFile.csv","a");
    fprintf(myfirstfile,"%4.2f, %3d, %3d, %3d, %3d, %1d\n",0.01,21,1,1,1,1);
    fclose(myfirstfile);
}
```

7 Conditional Statements & Loops

7.1 Conditional Statements

7.1.1 if statement

The if structure allows to check if certain conditions are fulfilled and to execute certain code only if the check returns 'true'.

```
int modelYear = 1950;
if (modelYear <= 1970){
   printf("Vintage!\n");
}
```

Note, that variables defined interior of the if construct are only present for said interior. The following code would not compile:

```
int a = 2;
if (a < 3){
   int b = 3;
}
printf("Because of this printf, the code will not compile... b=%d.\n",b);
```

7.1.2 else if statement

`else if` is only checked if the preceding condition(s) returned false, i.e. when all previous conditions are not met. Otherwise, it behaves the same as an `if` statement.

Generic Example Code

```
if (firstCondition)
{
   DoSomething();
}
else if (secondCondition)
{
   DoSomethingElse();
}
```

Compilable Code

```
if (a==5)
{
   cout << "a is equal to 5!\n";
}
else if (a==10)
{
   cout << "a is equal to 10!\n";
}
```

Many `if / else if / else if / ...` statements can be chained to form a complex condition check.

7.1.3 else statement

The `else` code block is executed if all other conditions are false.

Generic Example Code

```
if (firstCondition)
{
   DoSomething();
}
else if (secondCondition)
{
   DoSomethingElse();
}
else
{
   IfEverythingElseFails();
}
```

Compilable Code

```
int modelYear = 1950;
if (modelYear <= 1970)
{
   printf("Vintage!\n");
}
else if (modelYear <= 1960)
{
   printf("You will never see this message.\n");
}
else
{
   printf("Somewhat new!\n");
}
```

Remember: the first time one of the `if / else if / else` conditions - which are checked from top to bottom - returns true and its block `{}` is executed, the program jumps to the bottom of the entire `if` statement. No other conditions will be checked. This is different for the `switch` check presented below.

7.1.4 switch statement

Oftentimes, repeating the same equality check proves rather cumbersome. In these cases, it may be advantageous to use so-called switch-statements. These statements replace long chains of "if" and "else if" commands. The program jumps to a case according to a variable?s value and to default if no cases are a match.

Generic Example Code

```
switch (variable)
{
  case 0 : DoSomething();
   break;
  case 1 : DoSomething();
   break;
  default : DoSomethingElse();
}
```

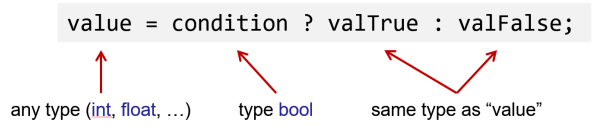
Compilable Code

```
unsigned int numberOfWorldCupsWon = 1;
printf("Country ");
switch(numberOfWorldCupsWon){
  case 4 : printf("Germany?\n");
           break;
  case 1 : printf("France?\n");
           break;
  case 0 : printf("Switzerland?\n");
           break;
  default : printf("... Never mind!\n");
}
```

Different to the `if` construct, the `switch` construct does not return after the first true comparison is reached, hence explaining the necessity of the `breaks`. Without an explicit break, program "falls through" to the next case. No break is needed for the `default` case (it's at the end anyway). The following code would also enter the `default` option even though case 2 is true, simply since there was no `break` set.

```
unsigned int numberOfBisous = 2;
printf("Are you ");
switch(numberOfBisous){
  case 0 : printf("German?\n");
  case 2 : printf("French?\n");
  case 3 : printf("Swiss?\n");
  default : printf("... Never mind!\n");
}
```

7.2 Conditional operators



The `?:` operator returns the value before `:` if the condition is true and otherwise the value after `:`.

7.3 Loops

In many cases we need to code programs that have repetitive tasks. This could be simply adding numbers on top of each other or more likely in the case of this course loop through several elements of a mesh or solving complex tensor calculations.

7.3.1 for loop

The for loop is the most common type of loop you will use. It provides a step-by-step declaration for implementing the desired action. These steps are as follows:

1. Defining the variable type and its initial value (`int counter = 10;`)
2. Defining a criterium that stops the loop (`counter >= 0;`)
3. Updating the defined control variable (`counter-`)
4. Executing the loop

```
for( int counter = 10; counter >= 0; counter--)
{
   cout << "The rocket lifts of in " << counter << " seconds" << endl;
}
```

Always pay attention to have a well defined and realistic termination criterion to prevent the formation of an endless-loop.

7.3.2 while-loop

A while loop essentially contains a conditional check as well as an execution body. If the conditional check is satisfied, the execution body is executed, followed by another check. This is repeated until the condition check is violated.

```
int workingHours = 0; // Declare + initialize loop variable
while (workingHours<9){
  // Execution body
  printf("Only %d hours left :)\n",9-workingHours);
  workingHours++; // incremental update of workingHours
}
printf("Stop. You have already worked for 9 hours!");
```

Both the `for` and `while` loop can be used equivalently. The loop variable `i` stays valid after the loop if declared outside of the braces. `for` loops are used for known length iteration and `while` loops to loop until a complex condition is met. The following code segment shows a comparison between the two loops:

for loop

```
int i;
for (i=0; i < 7; i++)
{
  cout <<"Value of i is:" << i << endl;
}
```

while - loop

```
int i=0;
while (i < 7)
{
  cout <<"Value of i is:" << i << endl;
  i++;
}
```

7.3.3 do while-loop

A `do-while` loop is similar to a while loop with the sole exception that it first conducts an execution and then a check.

```
int numberOfDirtyPlates = 10;
do {
  cout << numberOfDirtyPlates << " still need to be cleaned " << endl;
  numberOfDirtyPlates--;
} while (numberOfDirtyPlates > 0);
```

8 Functions

Functions have a very diversive spectrum of advantages to offer. In all generality, they provide a mean to define a set of instructions to one method, so as to be able to call said collection of instructions multiple times without having to retype them. This obviously allows for a much nicer code structure, but can also lead to performance in speed, e.g. with respect to compile time. In Ch. 13, arguably the most powerful tool in C++ will be introduced, namely classes. Within said classes, functions can be defined so as to "only be visible within said class" or "globally visible". This access categorization additionally adds structure and even to some extend security to the code.

In a first overview, the general syntax adopted by a function comprises the following components:

- *function name*: name of the function
- *function body*: statements to be executed
- *argument*: variable whose value is passed into the function body from the outside - can be empty
- *argument type*: type of the argument if existent
- *return value*: value that is passed to the outside after function call - inexistent in the case of so-called *void* functions
- *return type*: type of the return value - N.A. in the case of so-called *void* functions

The general syntax to define a function adopts the following structure

```
returnType
functionName(argumentType0 argumentName0,argumentType1 argumentName1){
             /* ------------------Argument List ------------------------*/
  returnType returnValue; //
  Command0();             // Function
  Command1();             //   Body
  // ...                  //

  return returnValue;
}
```

The general way to exert a function call from somewhere else (the definition of what *somewhere else* means will be defined in Ch. 13) can be summarized as follows:

```
  argumentType0 hereIsYour0thArgument;
  argumentType1 noArgumentRequiresTheSameNameAsWrittenInTheFunction

  returnType letsSeeWhatTheFunctionGivesUs
    = functionName(hereIsYour0thArgument,↩
        noArgumentRequiresTheSameNameAsWrittenInTheFunction);
```

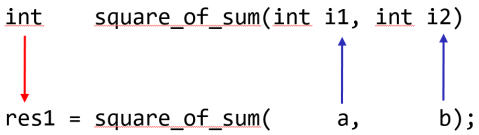
Let's consider a more "concrete" example: The following code shows a very simple function that takes the minor and major radius of an ellipse and returns its area. We can call it from any other function, such as for example main():

```
#include<stdio.h>
using namespace std;

double
calculateTheAreaOfAnEllipse(double minorRadius, double majorRadius){
  double area = 3.1415 * minorRadius * majorRadius;
  return area;
}
```

```
9
10 int main(){
11   double radiusMinor = 1.0;
12   double radiusMajor = 2.0;
13
14   double areaOfEllipse
15     = calculateTheAreaOfAnEllipse(radiusMinor,radiusMajor);
16   printf("The area of the ellipse is %7.4f.\n",areaOfEllipse);
17
18   return 0;
19 }
```

Note that the argument that is passed to the function in the main file and the argument in the function itself do not need to have the same name



Const

Oftentimes we want certain parameters to stay constant throughout the function call and prevent accidental changes to said variable (*errare humanum est* - to err is human). The qualifier *const* allows to achieve exactly this.

Out of the following two examples, only the first one would compile.

Example 1:

```
1 double
2 add7ToArgumentAndReturn(double pristineData){
3    pristineData += 7;
4    return pristineData;
5 }
```

Example 2:

```
1 double
2 add7ToArgumentAndReturn(const double pristineData){
3    pristineData += 7;
4    return pristineData;
5 }
```

9 Static vs. Dynamic Memory Allocation

Of course, when initializing objects, we automatically "reserve" or allocate some part of the memory available to us. The memory of a computer can be seen as a large collection of consecutive blocks (1 *byte* equates 8 *bits*). A bool reserves 1 byte, a float 4 bytes and a double 8 bytes. The way in which this is done can be one of two:

- **Static allocation:** the memory is pre-allocated at compiling time, i.e. you need to know how much space exactly said object is going to take by the time you are compiling your program. To this end, the initialization in all our examples were of this type. **Caution**: Static allocation occurs on the memory's heap. Oftentimes, the heap is of limited size (much smaller than the total available memory), in which case one may have to divert to dynamic allocation. In such an event, there is oftentimes no warning thrown at compile time, but instead a *segmentation fault* at run time.

- **Dynamic allocation:** the memory is allocated at run-time. This is particularly useful in the framework of arrays and vectors introduced shortly, when the total size of a series of elements is *a priori* unknown. **Caution**: Unlike static allocation, dynamic allocation requires memory control, in that as soon as a variable is 'of no use anymore', it has to be deleted off the memory.

10 Pointers & References

In the previous section, we introduced memory allocation. Every memory that is reserved for an int, double or any other object is endowed with a unique address. *Pointers* are objects that can safe exactly said addresses. An integer pointer safes the address to an integer, a double pointer safes the address to a double and a someGenericClass pointer safes the address to an object of someGenericClass. The following code and comments illustrate the definition of pointers, referencing as well as dereferencing.

```
#include<stdio.h>
using namespace std;

int main(){
  int   someActualInteger ; // This is an actual integer
  int * someIntegerPointer; // This is a pointer to an integer
                            // i.e. an address to an integer

  someIntegerPointer = &someActualInteger;  // The &-operator asks
                                            // for the address of
                                            // someActualInteger.

  someActualInteger = 5;
  printf("someActualInteger has the value: %2d.\n",someActualInteger);
  printf("which can also be called via   : %2d.\n",*someIntegerPointer);
  // Note: the * has two functions ...
  //       1. ... when in front of a class, it means, we are defining a pointer
  //       2. ... if it stands in front of any pointer, it means "dereference", i.e.
  //              access what is written in the memory that the pointer(=address)
  //              points to.

  (*someIntegerPointer)++;

  printf("someActualInteger has the value: %2d.\n",someActualInteger);
  printf("which can also be called via   : %2d.\n",*someIntegerPointer);

  return 0;
}
```

Caution: when defining multiple pointers in one line, always repeat the *: *int *a, *b, *c;*

10.1 Passing by pointer, passing by reference

In Chapter 8 we considered passing and returning objects of different types to and from functions. To this end, these objects that were passed over to some functions were implicitly copied onto another instance and then passed on. Since this may seem abstract at first, let's take a look at a concrete example:

```
int multiplyBy10(int a){
  a *= 10;
  return a;
}
int main(){
  int b = 10;
  int c = multiplyBy10(b);
  printf("The final values are: b=%d | c=%d.\n",b,c);
} // The output will be b = 10 and c = 100
```

We observe immediately that while the function argument b should be multiplied by 10 inside the function, it still bears the value of 10 after the function has returned. As explained before, the reason for this is that a copy of the original argument is implicitly created, which has no influence whatsoever on the original b. If however we want to prevent this creation of a copy, but instead somehow handle the original argument, there are two ways to achieve this:

- **Pass by reference:** If the object itself is supposed to be passed on, one can use the ampersand symbol &. The following code illustrates this concept:

```
int multiplyBy10(int & a){
  a *= 10;
  return a;
}
int main(){
  int b = 10;
  int c = multiplyBy10(b);
  printf("The final values are: b=%d | c=%d.\n",b,c);
} // The output will be b = 100 and c = 100
```

- **Pass by pointer:** If the address to the memory space - at which the object is located - is supposed to be passed on, one can use *. "Addresses", which in C++ are more commonly referred to as pointers, are defined by the star symbol *. 'int * a' indicates that a is an address/pointer, pointing towards an integer, which can be called via '*a'. 'char * b' indicates that b contains the memory address of some char which itself can again be called by '*b'. It is completely normal if this double-functionality of '*' may seem rather confusing to you. But the rule is strict and there are no exceptions:
  - The syntax '<type> * <nameOfPointer>' indicates, that we treat a <type>-address/pointer with name <nameOfPointer>.
  - In all other cases, '* <nameOfPointer>' indicates that we access the address stored in <nameOfPointer> to obtain what is stored in the memory bank with said address.

  Applying what we learnt to our previous examples leads to the following code

```
int multiplyBy10(int * a){  // Here, int * indicates the type - an integer-pointer
  *a *= 10;   // In these two lines *a indicates dereferencing, i.e. "follow"
  return *a;  // address a (-> type int *) and consider what is stored there
              // (-> type int).
}
int main(){
  int b = 10;
  int c = multiplyBy10(&b);
  printf("The final values are: b=%d | c=%d.\n",b,c);
} // The output will be b = 100 and c = 100
```

11 Arrays & Vectors

11.1 std::array (static container)

Until now we have only worked with simple types like int, char, float, double, bool. An array as provided by the std-library (std::array) is a fixed sized data structure working as a static memory allocation capable of storing multiple values of the same type. All entries in the array need to be of the same type and stored in separate cells. Every cell of the array can be accessed individually using a distinctive index. Once you define the array, it cannot be resized. This size must be specified at compile time. Array indexing is zero-based, which means that indices (subscripts) go from 0 to "size-1" (where size is the number of elements). Both array and vector use the bracket operator (e.g [ ]) to access elements. The first element is vectorA[0], the third element is vectorA[2]. C++ does in general not check whether your index is valid at compilation time and segmentation faults may occur at runtime. The definition consists of three elements: Data type of elements, number of elements and the name of the array. There are a couple of different options on how to define and initialize an array.

Method 1: (Explicitly)

```
array<ObjectType, N_Objects> anyNameYouWant;
anyNameYouWant[0] = Object1;
anyNameYouWant[1] = Object2;
```

Method 2: (At definition)

```
array<ObjectType, N_Objects> anyNameYouWant = {Object1, Object2 ..., Object_N};
```

Method 3: (Fill)

```
array<ObjectType, N_Objects> anyNameYouWant = {Object1, Object2 ..., Object_N};
ObjectType someSampleObject;
anyNameYouWant.fill(someSampleObject);
```

The following is an example on how to initialize an array and fill the cells with values and how to access individual elements using square brackets:

```
array<double,4> randomArrrayName;
randomArrayName[0] = 0;
randomArrayName[1] = randomArrayName[0] + 1;
randomArrayName[2] = randomArrayName[1] + 2;
double iWantToAddTwoNumbersFromArray = randomArrayName[0] + randomArrayName[1];
```

Caution: It is of central importance that the number of elements is constant when defining the array:

```
int       i=10;
const int j=10;

float myFloats [i]; // Invalid; must be sure about size of array at compile time.
float myFloats2[j]; // This is okay.
```

11.2 Vector (dynamic container)

A vector is a variable sized data structure working as a dynamic memory allocation. Its initial size is 0. It's size can be changed during runtime. It can be seen as a dynamic array that does not need a predefined length. The definition of a vector via the empty constructor is as follows

```
vector<ObjectType> anyNameYouWant;
```

Alternatively, one can pre-define a non-negative size and optionally even pass an object which is to be saved in all entries of the vector:

```
ObjectType someObject;
vector<ObjectType> someVector(numberOfInitialEntries,someObject);
vector<ObjectType> someEmptyVector(0);
```

The following is an example on how to initialize a vector and fill it with values:

```
vector<double> vectorOfDoubles(0);
vectorOfDoubles.push_back(1);
vectorOfDoubles[0] = 2; // this reassigns the value of the first entry in the vector
```

Initially the vector always has the length zero. Only by pushing back values one can expand the size of the vector. This is also why the following command can not work as the vector has size 0 and you are trying to assign 1 to the first entry of the vector, which doesn't exist.

```
vector<int> vectorOfInts;
vectorOfInts[0] = 1;
```

12 Eigen library

The eigen library can be used for linear algebra, matrix and vector calculations as well as geometrical transformations and numerical solvers. A good reference for the Eigen library is: `http://eigen.tuxfamily.org/dox/AsciiQuickReference.txt`. The eigen library provides static and dynamic vectors and matrices depending on how you define them. Uses the parenthesis operator (e.g. ());

Static vectors and matrices

The size of the vector or matrix is predefined and the memory needed to save the tensor is pre-allocated

```
Matrix<StandardDataType,numRows,numCols> staticMatrix;
Matrix<StandardDataType,numRows,1> effectiveStaticVector;
```

Dynamic vectors and matrices

Dynamic matrices

```
unsigned int numRows = 10;
unsigned int numCols = 10;

//Method 1 for matrices:
MatrixXd dynamicMatrix1;
dynamicMatrix1.resize(numRows,numCols);
dynamicMatrix1.fill(0);

//Method 2 for matrices:
MatrixXd dynamicMatrix2(numRows,numCols);
dynamicMatrix2.fill(0);
```

Dynamic vectors

```
unsigned int numRows = 10;

//Method 1 for Vectors:
VectorXd dynamicVector1;
dynamicVector1.resize(numRows);
dynamicVector1.fill(0);

//Method 2 for Vectors:
VectorXd dynamicVector2(numRows);
dynamicVector2.fill(0);
```

The following example shows how one can access different cells of the matrix:

```
MatrixXd dynamicMatrix(10,10);
dynamicMatrix.fill(0);
dynamicMatrix(0,0) = 1;
dynamicMatrix(3,2) = dynamicMatrix(0,0);
```

The Eigen library has several implemented tensor computations pre-defined. The following code segment shows five of these commands that are frequently used. For further definitions please refer to the website of Eigen.

```
MatrixXd generalMatrix(10,10);
generalMatrix.fill(0);
// 1) Gives information about the number of rows
      unsigned int matrixRows = generalMatrix.rows();
// 2) Gives information about the number of columns
      unsigned int matrixCols = generalMatrix.cols();
// 3) Transpose the matrix and assign it to another matrix
      MatrixXd transposedMatrix = generalMatrix.transpose();
// 4) solve a linear system Ax = b
      MatrixXd matrixA(10,10);
      matrixA.fill(0); // always good practice to do this
      VectorXd vectorb(10);
      vectorb.fill(0);
      VectorXd x = matrixA.lu().solve(vectorb);
// 5) add the numbers from each vector into one sum
      double sum = 0
      unsigned int numberOfVectorTerms = 10;
      vector<double> firstVectorOfDoubles;
      vector<double> secondVectorOfDoubles;
      for (unsigned int index = 0; index < numberOfVectorTerms; index++){
            sum += firstVectorOfDoubles[index] + secondVectorOfDoubles[index];
      }
```

13 Classes

A class in C++ can be seen as a container that stores both data and functions in one package. Classes can be defined in libraries or the main file. To access the class, the library it is defined in needs to be included. Member variable and functions of a class can be categorized according to access rights from outside the class

- **public:** accessible from outside the class.
- **private:** cannot be accessed from outside the class. Only methods inside the class can access these members. **Note:** by **default** access to members of a class is private
- **protected:** equivalent to private with the additional feature that classes inheriting from the class where the protected member is defined, can also access it. Since we barely work with inheritance in our code, you can forget about this third option.

Note: As mentioned before, by (our) convention, member variable names commence by an underscore.

13.1 Object-oriented languages

C++ just like Java is an **object oriented language**. That being said, we structure the code in a way, so that instances of the same or similar type using/providing the same functionalities can be described by one **class** with different settings for the member variables. These instances are referred to as **objects**. Since this might be the first time you work with an object-oriented language, let's rephrase this one more time: Classes collect a set of member functions and variables. An object of a class endows all these member variables and sets specific values for them.

How is that useful you may ask. Working on problems in Solid Mechanics, in which we work with models on an every-day basis, many of which share many functionalities and structures simply with different parameter values, suggests the definition of such models as classes. We could for example define the class `MaterialModelNeoHookean` and define two instances for aluminum and magnesium.

```
class MaterialModelNeoHookean {

  public:
  MaterialModelNeoHookean(double youngsModulus,
                          double poissonsRatio){ // Constructor
    _youngsModulus = youngsModulus;
    _poissonsRatio = poissonsRatio;
  }

  // A public function callable from the outside
  double computeEnergy(/* Some Input */){
    double energy = 0.0;
    /* Some Instructions */
    return energy;
  }

  private:
  // Two private members, inaccessible from the outside
  double _youngsModulus;
  double _poissonsRatio;

};

int main(){
  MaterialModelNeoHookean neoHookeanAl(68.9,0.33);
  MaterialModelNeoHookean neoHookeanMg(45.0,0.35);
  return 0;
}
```

13.2 Constructor

When an object of a class is created, one explicitly or implicitly calls a constructor. Constructors allow to define both public as well as private members when initializing the object. While this is a nice feature for non-constant members, it is vital for `const` members, which are defined the following way:

```
class MaterialModelNeoHookean {

  public:
  MaterialModelNeoHookean(double youngsModulus,
                          double poissonsRatio): // Notice the colon!
    _youngsModulus(youngsModulus){ // Valid!
    _poissonsRatio = 3.3;          // Invalid !
  }

  private:
  // Two private members, inaccessible from the outside
  const double _youngsModulus;
  const double _poissonsRatio;

};
```

Note: Even if the invalid line of code was removed, it would not compile, simply because a constant member was not defined. A compilable constructor would look as follows in this case:

```
MaterialModelNeoHookean(double youngsModulus,
                        double poissonsRatio):
  _youngsModulus(youngsModulus),
  _poissonsRatio(poissonsRatio){}
}
```

13.3 The object as an implicit argument to functions

In Ch. 8, we stressed the importance of the *const* qualifier. It allowed to avoid to some extend human error in that variables which were not intended to be changed could be protected from any accidental change via said qualifier. Apart from variables defined inside a function or input variables, one can also ensure that the object - through which the function of interest is exerted - is not changed. By the *object is not changed*, we mean that none of the member variable is subjected to changes. The following examples illustrates this:

```
double
computeEnergy(Matrix<double,9,1> displacementGradient) const {// The last const here
                                                              // signalizes that the
                                                              // object which calls it
                                                              // does not change
  double energy = 0.0; // We clearly do not change any member variables
  return energy;
}
```

To get a better understanding of this concept, it is important to know that if a function is called through an object, i.e. <objectName>.<functionName>(<listOfArgument>), then <objectName> itself is implicitely an argument to the function and the const tells the outside, that the object as an argument to that function will not change. That being said, the following code would yield an error at compilation time, which can be fixed by reinserting the *const* qualifier as shown above for *computeEnergy*.

```
class MaterialModelNeoHookean {

  public:
  MaterialModelNeoHookean(double youngsModulus,
```

```
5                           double poissonsRatio):
6       _youngsModulus(youngsModulus),
7       _poissonsRatio(poissonsRatio){
8   }
9
10  // A public function callable from the outside
11  double
12  computeEnergy(double displacementGradient) {// The last const here
13                                              // signalizes that the
14                                              // object which calls
15                                              // the function does not change
16    double energy = 0.0; // We clearly do not change any member variables
17    return energy;
18  }
19
20  private:
21
22  // Two private members, inaccessible from the outside
23  const double _youngsModulus;
24  const double _poissonsRatio;
25
26 };
27
28 int main(){
29
30  const MaterialModelNeoHookean neoHookeanAl(68.9,0.33);
31  neoHookeanAl.computeEnergy(0.0);
32
33  return 0;
34 }
```

14 Templates

Oftentimes, when defining classes, functions, etc. one likes to keep the choice of data type for return types, input parameters, member variables, etc. open. Templating allows exactly that. For example, consider the infinity norm for a vector. The infinity norm is defined for both integers, doubles, etc. and we could very well in an old-fashioned style define one function each for *vector<double>* and *vector<int>*:

```
int
infinityNorm(vector<int> inputVector){
  int largestValue = inputVector[0]; // just setting a value
  for (unsigned int index = 0; index < inputVector.size(); index++){
    if (inputVector[index] > largestValue){
      largestValue = inputVector[index];
    }
  }
  return largestValue;
};
double
infinityNorm(vector<double> inputVector){
  double largestValue = inputVector[0]; // just setting a value
  for (unsigned int index = 0; index < inputVector.size(); index++){
    if (inputVector[index] > largestValue){
      largestValue = inputVector[index];
    }
  }
  return largestValue;
};
```

Templating allows to circumvent repeating this issue of having two functions with the exact same structure defined, just because they differ in one data type. The following code shows how we can efficiently reduce the number of functions by introducing templating the function infinityNorm by some - at this point - unknown class InputType, that only needs to be known at compile time

```
template <class InputType>
InputType
infinityNorm(vector<InputType> inputVector){
  InputType largestValue = inputVector[0];
  for (unsigned int index = 0; index < inputVector.size(); index++){
    if (inputVector[index] > largestValue){
      largestValue = inputVector[index];
    }
  }
  return largestValue;
}
```

The specification can at times be implicitly deduced by the compiler, as shown in this example:

```
int main(){
  unsigned int numberOfValuesToInitializeVectorWith = 4;
  int          valueToFillIntVectorWith             = 1;
  double       valueToFillDoubleVectorWith          = M_PI;

  vector<int> inputVectorInt(numberOfValuesToInitializeVectorWith,
                             valueToFillIntVectorWith           );
  vector<double> inputVectorDouble(numberOfValuesToInitializeVectorWith,
                                   valueToFillDoubleVectorWith         );
  // Specify template type explicitly
  int    largestValueInVectorSpec       = infinityNorm<int>   (inputVectorInt);
  double largestValueInDoubleVectorSpec = infinityNorm<double>(inputVectorDouble);
  // Specify template type implicitly
  int    largestValueInVectorNonSpec       = infinityNorm(inputVectorInt);
  double largestValueInDoubleVectorNonSpec = infinityNorm(inputVectorDouble);
  return 0;
}
```

15 Typedef & Typename

15.1 typedef - Keeping things short / In der Kürze liegt die Würze

Shorthand so you don't have to write out really long expressions all the time. You can also reach into classes to use their typdefs (as long as they are public.) The following example shows several typedefed matrices that are defined in MaterialModelBar1D.h and then reused in Main.cc

MaterialModelBar1D.h Matrix<double, 1, 1>

```
class MaterialModel1DBar {

  public:
  typedef Matrix<double, 1, 1> DisplacementGradient;
  typedef Matrix<double, 1, 1> Strain;
  typedef Matrix<double, 1, 1> Stress;
  typedef Matrix<double, 1, 1> TangentMatrix;

  \\ ...
};
```

15.2 typename

So far, so good. If you now want to typedef with respect to a template parameter, C++ wants to know that and the keyword here is `typename`

Main.cc

```
template<class MaterialModel>
class Element{
  typedef typename MaterialModel::DisplacementGradient     DisplacementGradient;
  typedef typename MaterialModel::Strain                   Strain;
  typedef typename MaterialModel::Stress                   Stress;
  typedef typename MaterialModel::TangentMatrix            TangentMatrix;

  \\ ...
};
```

There are other uses of typename, but in the interest of briefness, we shall redirect the interested reader to `http://pages.cs.wisc.edu/~driscoll/typename.html`

16 Namespace

Namespaces are used to define the scope of where functions, classes, structs, etc reside in. This scope helps you distinguish between two Element classes that may have the same name and take in the same arguments with the same data types such as the small strain bar element and the small strain with damage bar element. Both have the same exact names and constructors, but the internal workings are different. Also usually libraries will have their own namespace such as std:: or Eigen::

```
namespace Elements{
  namespace SmallStrain{
    class BarElement{
      BarElement(Vector<double,2,1> nodePositions,
                 double stiffness)
      // filled with public variables, constructor, methods, private variables
    }
  }
  namespace SmallStrainWithDamage{
    class BarElement{
      BarElement(Vector<double,2,1> nodePositions,
                 double stiffness)
      // filled with public variables, constructor, methods, private variables
    }
  }
  namespace FiniteStrain{
    class BarElement{
      BarElement(Vector<double,2,1> nodePositions,
                 double stiffness)
      // filled with public variables, constructor, methods, private variables
    }
  }
}
```