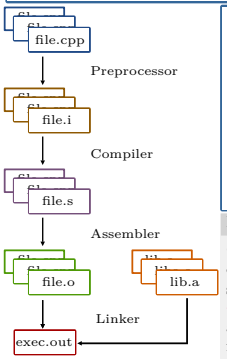
Compiling a C++ Program

Definition 1.1 Source Code file.c/cpp: Is the highest level, human readable form of a computer program that can be translated into machine language with the help of a compiler?? .

Definition 1.2 Compiler: Is a computer program or set of programs, that may include a preprocessorsection 1, an assemblersection 2 and or linkersection 3. The task of a compiler is to translates source code of a high-level programming language, e.g. C++ into a low level language e.g. assembly language?? or directly into machine language/binary code.



Definition 1.3 file.asm/s **Assembly Language (asm):** Is a low-level programming language that can be translated one-to-one from a form readable text (not just ones and zeros) to machine code.
In contrast to high level programming languages each assembly language is (usually) computer architecture specific.

Note: creating Assembly code

Compilers usually have a flag in order to compile source files into Assembly code i.e. when supplying the -S option gcc does not run the linker, assembler and creates an assembly file file.s.

1. The Preprocessor *.i

Definition 1.4 The Preprocessor: Is a program that performs preliminary operations on C and CPP source files before they are passed to the compiler.
It can be used to conditionally compile code, insert files, specify compile-time-error messages and apply machine specific rules to sections of codes.
Preprocessor commands start with a hashtag #.

1.1. Defining Macros #define

Definition 1.5 Macro: Is a fragment of code which has been given an name, that can be used as an alias for that piece of code.

Definition 1.1 define: lets us define a macro:

```
#define alias macro
```

the macros are then simply pasted into the code by the preprocessor.

Definition 1.2 undefine: lets us undefine a macro:

```
#undefine alias
```

Note

we can also (un)define macros on the command line:

```
compiler -Dmacroname1 -Dmacroname2=value
compiler -Umacroname1 -Umacroname2
```

1.1.1. Predefined Macros __Macro__

There exist many predefined macros s.a. __FILE__, __TIME__ that start and end with two underscores:

```
__Macro__
```

1.2. Conditional Statements

1.2.1. #if...#elif...#endif

Allows conditional expressions:

```
#if 3>2
  somthing
#endif
```

1.2.2. #defined,#ifdef,#ifndef...#endif

Allows us to check if certain macros are defined see example section 5

Definition 1.3 #ifdef: allows to check if a *single* macros is defined:

```
#ifdef(__MACRO__)
  somthing
#endif
```

Definition 1.4 #defined: allows to check if a *single* or macro is defined but allows also to do compound comparisons:

```
#if !defined(__MACRO1__) && defined(__MACRO2__)
  somthing
#endif
```

1.3. Includes

allows us to include another (or many) source file(s) into our source file by pasting the entire content of the other file into our file, at the point where we call that command.

Definition 1.5 Including Own Files:

```
#include filepath.(h/hpp)
```

Notes

- We may also specify default include directories s.t. we do not need to type the path: `compiler -I dir`
- include files before parsing directly on the command line `compiler -include file`

Definition 1.6 Including Standard Files: The preprocessor searches for header files??

```
add header file somewhere
```

(usually standard library header files) in an implementation dependent manner in some predefined directories. Those libraries can then simply be included by:

```
#include <lib>
```

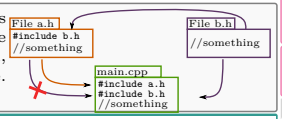
Note

You can find the predefined include directories by using:

C	`gcc -xc -E -v -`
CPP	`gcc -xc++ -E -v -`

1.3.1. Include Guards

Helps to avoid including files multiple times when multiple files object files are linked, that include the same file. This may even lead to cycles.



Definition 1.7 FILE_(H/PP)**:** This can simply be done by defining a new macro if the file has not been included yet:

```
#ifndef FILE_HPP
#def FILE_HPP
  // file content
#endif
```

We could define any name we want as macro, as long as it is unique but the convention is to use the uppercase filename plus the uppercase file extension separated by an underscore.

1.4. Errors

Definition 1.8 #error message: Allows to issue error message at compile time i.e. if certain conditions are not met.

1.5. Asserts #include <cassert>

Definition 1.9 Assert: Asserts can be used in order to check *preconditions*??,*postconditions*?? and *invariantes*??:

```
add pre and postconditions somewhere
```

```
assert(condition)
```

the condition is checked at *runtime* and if it does not hold the program will terminate with an error message.

Note: error message

We may change the error message by using:

```
assert(cond && "message")
```

1.5.1. Disabling Assertions #define NDEBUG

Assertions may be disabled/deleted by defining the macros NDEBUG, which stands for no-debug an may be defined at the command line:

```
compiler -DNDEBUG
```

this is also the reason why assert() is a macro and not a function!

2. The Assembler *.asm/*.s

Definition 1.6 Assembler: Assembly language gets converted into objectsection 2 files by a program called assembler.

Definition 1.7 Object File file.o: The assembler creates an object file for each source file. The object file may contain executable code as well as information for the linker section 3.

3. The Linker *.o

Definition 1.8 Linker: The Linker is a computer program that takes one or many object files?? and combines them into one single executable file, library, or another object filesection 2.

Note: System Libraries

Are usually linked by default.

4. Running The Compiler

4.1. Verbosity/Warnings -W

Warnings report other unusual conditions in your code that may indicate a problem, although compilation can (and does) proceed.

```
gcc [-Q] --help=warnings
```

-Q: if the -Q options appear before a --help=option, ten the help indicates which of the options are enabled or disabled.

-Wall: Enables a lot of different warnings that "most people" agree on.

-Wextra: enables some extra warning flags that are not enabled by -Wall

-Wpedantic: Issue all the warnings demanded by strict ISO C and ISO C++section 2.

Note

With this option certain by GNU supported extensions and traditional C and C++ features are rejected.

4.2. Important Options

-o filename**:** Specifies the filename for the output.
This also holds for running only certain steps section 3.

-std=standard**:** sets the C/C++ standard according to which a compilation unit should be compiled.

Note

The default is compiler and version dependent but can be deduced i.e. by running the debugger.

4.3. Running Only Certain Steps

-E: runs only the pre-process to produce file.i.

-S: runs only the pre-process and initial compilation in order to produce assembly files?? file.s.

-C: runs only the pre-process, initial compilation and the assembler i.e. do not run the linker in order to produce object filessection 2 file.o.

4.3.1. GCC/G++ gcc/g++

4.3.2. Clang clang

5. Examples

Example 1.1 Example: Checking for compiler versions:

```
#if defined(DEBUG)
  somthing
#endif
```

Example 1.2 Example: Checking for compiler versions:

```
#if !defined(__GNUC__)
  somthing
#elif __GNUC__ <= 2 && __GNUC_MINOR < 95
  somthing
#elif __GNUC__ > 2
  somthing
#endif
```

CMake

Often Makefiles become overly complex – particular when building projects that have many sub directories, projects that deploy to many platforms or projects that build and use libraries.

Definition 2.1 CMake: Is a crossplatform generator that automatically generates Makefiles. CMake projects consists of `CMakeLists.txt` files that describe your project.

Definition 2.2 Target: A CMake-based buildsystem is organized as a set of high-level *logical targets*. Each target corresponds to an executable or library, or is a custom target containing custom commands and will result into a *Makefile target*

C++ Basic Concepts

Definition 2.3 ANSI-/ISO-/and Standard-C/C++**:** Are successive standards for the C/C++ programming language published by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Software developers writing in C/C++ are encouraged to conform to the standards, as doing so helps portability between compilers.

1. Programming Paradigms

Definition 2.4 Procedural Programming**:** Is a programming paradigm based on procedures, also known as routines, or functions. Any given procedure might be called at any point during a programs execution, including by other procedures as well.

Definition 2.5 Modular Programming**:** Modular programming involves dividing our objects into modules of common characteristics and then piplining the modules to obtain the defined output. Modules consit of an interface and an implementation. The interface describes the functionality of the module to the outer world and explains how other modules can make use of it.
E.g. separating files, Libraries.

Definition 2.6 Object Oriented Programming (OOP)**:** Builds on modular programming but focuses rather on data itself than algorithms.
Programs are organized as objects: data structures consisting of data fields and member functions.

Definition 2.7 Generic Programming**:** Style of programming in which algorithms are written in terms of types "to be specified later" that are then instantiated when needed for specific types provided as parameters.
E.g. Templates

1.1. Contract Programming

Definition 2.8 Pre-Conditions**:** A condition that needs to be satisfied prior to the execution of a function.

Definition 2.9 Post-Conditions**:** A condition that is ensured to be held after the execution of a function.

Definition 2.10 Invariant**:** A condition that needs to be satisfied prior to the execution of a function and is ensured to be held after the execution of a function.

Definition 2.11 Contract**:** Set of preconditions, postconditions and invariants associated to a function.

2. The Translation Unit

3. Deceleration Paradigm

3.1. Types Of Declarations
3.2. Forward Declarations
3.3. The One Definition Rule

Definition 2.12 Names/Identifiers**:**

Definition 2.13 Deceleration**:**

Definition 2.14 Definition**:**

Note

Definition 2.15 Translation Unit**:**

Definition 2.16 One Definition Rule**:**

3.4. The Scope

4. Linkage

4.1. Static
4.2. Extern

Libraries

1. Static Libaries/Archieves *.lib/so/ar

Static Libraries/Achieves are nothing else but a couple of object files *.o packed together, that can be linked into your program.

1.1. Creating Static Libraries

1.1.1. Creating an Achieve

(1) Creating an achieve from object files *.o

```
ar [options] libname.a file1.o file2.o ...
```

(2) Add index/table of contents (not needed on all platforms)

```
ranlib libname.a
```

1.1.2. Achiever Options ar -option

-c: Create an archive/library and do not warn if the library has to be created.

-r: Add object files *.o to the library and replace any existing files with the same name.

-u: Replace the member only if the modification time of the file member is more recent then the time of the file in the archive.

Note

A good combination to use is ruc.

Note: ranlib

ranlib generates and adds an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

1.2. Compiling&linking achieves into our Program

linking the library/archive into our program

```
gcc|g++|... -static main.cpp [-Ldir]  -llibname.a
```

-L: Specify the path from which to include the library

2. Dynamic Libaries *.ddl/*.a

3. Header Only Libaries *.h

Classes

Definition 3.1 Classes: are boiler plates for ..

```
class MyClass [:[visibility_mode] BaseClass]{
  public:
  MyClass()                    ← Constructor
  ~MyClass()                   ← Destructor
     ...
  protected:
     ...
  private:
     ...
  friend:
     ...
}
```

Class names use capital letters for class names.

Member Variables use underscore suffixes for member variables `var_`

4. Constructors

4.1. Default Constructor

Definition 3.2 Default Constructor:

todo

Corollary 3.1 Default Construction of Members:

- Any member of type `class`, `struct` that is not initialized in the initialize list will be default constructed.
- For a C-style array each element that is a `class`, `struct` will be default constructed.
- Any built-in/primitive data type will *not be initialized*!
  ⇒ need explicit initialization to avoid undefined behavior!
- Objects with static or thread storage duration are zero-initialized

Explanation 3.1 (
Why do we need to initialize built in types explicitly?).
Built in types do not have a true default constructor as it would seem:

```
int x(5);              Looks and Behaves like a constructor
int z = int();  Looks and Behaves like a constructor (0 init)
int y();          BUT is actually a function declaration!
```

5. Member Initializer Lists

6. Member Functions/Methods

6.1. Constant Methods

Definition 3.3 Constant Methods: are functions that are *not allowed* to change any *class members*.

```
Type fu(...) const;                    declaration
Type fu(...) const {...};              definition
```

6.2. Constant Method Arguments

Definition 3.4 Constant Method Arguments: tells the user of a function that the argument(s) with `const` will not and cannot be changed inside the function:

```
Type fu(const var [,...]) [const];     declaration
```

7. Static Members

Definition 3.5 Static Member/Variable: Are variables that exist once *per class* (and not per object):

```
MyClass::myVar [ = val;]                          access
```

```
class Foo                                    deceleration
{                                                  .hpp
  private:
     static type var;
};
```

1. The initialization of static members is not allowed inside the header file (`.hpp`) as it would violate the one definition rulesection 3. In other words: the linker would not know which definition to choose if multiple files would include the header file.
2. ISO C++ forbids in-class initialization of `non-const static` member ⇒ static members must be declared inside the class but initialized outside the class.

```
Foo::var = [ = val;]                      definition [.cpp]
```

Explanation 3.2. *That's because the declaration is a description of how memory is to be allocated, but it doesn't allocate memory. There will only be one instance of the static variable, which is basically kind of a global variable.*

understand why inclass initialization is not possible

Note

- Static and thread storage duration objects without value initialize are zero initialized:
  `type Foo::var;` `definition [.cpp]`
  is actually a definition (and not a declaration) initialized to zero.
- Static members are not subject to access controls
  ⇒ can even initialize the value if it is declared as private.

8. Inheritance

8.1. Virtual

Exception Handling

Murphy's Law's:
- "It is impossible to make anything foolproof because fools are so ingenious."
- "If anything simply cannot go wrong, it will anyway."

Simply using if statement has the drawback that errors are not separated from the logical code.
⇒ code becomes hard to read and maintain. In addition to that we do not need to write lots of unreadable if-statements in a sequence, as errors can be composed to *error objects* of certain types.

Question: when should we use exceptions?

1. When trying to catch *other peoples* errors
2. Use exceptions when trying to catch problems that might occur
3. Do not use exception to address problems that would mean a serious *flaw/invariants* in your code.
   To check things that actually never occur rather use *assertions*??.

1. Try & Catch

Definition 4.1 Try & Catch: is a mechanism to react on errors. Every try block is followed by at least a catch block to handle possible errors.

```
try { ...
} catch(ExceptionType1 Identifier){
    handle_exception
}[catch(ExceptionType2 Identifier){
    handle_exception
}...]
```

the *catch block* catches errors according to the ExceptionType of the `throw` obj, which can be any arbitrary type.

Corollary 4.1 Catch All Others: If we want to catch all errors that have not yet been catched we can make use of:

```
...
} catch(...){
    handle_all_other_exceptions
}
```

this is useful if an error is thrown that we did not anticipate.

2. Throw

Definition 4.2 Throw: Allows us to throw/create errors if certain undesirable conditions happen:

```
throw someObj;
```

someObj can be any arbitrary object even `int`,`double`,...

Corollary 4.2 Consequences of Throw:
1. The normal execution of the program stops
2. The call stack is unwound, the functions are excited, all local objects are destroyed
3. If the exception is `thrown` within a try block and the exception can be `catched` it will be handled int he corresponding catch block.
4. Afterward execution will be resumed after the try block. Otherwise the program terminates.

Note

If we want to throw 42.4 and catch it we need to catch a `double` (standard in C++; not float).

2.1. Rethrow

Definition 4.3 Rethrow: If an exception cannot be handled fully in a catch block we may rethrow the exception:

```
...
} catch(ExceptionType1 Identifier){
    fail_to_handle_exceptions
    throw;                          rethrow exception
}
```

2.2. Throw Declarations

Definition 4.4 Throw Declarations: allows us indicate which functions might be thrown.

```
rType myFun(args) throw(Type1 [,Type2,...])
{
  ...
}
```

Warning: The compiler does not even check if a throw is missing. **But** it complains if you try to throw something not declared.

Note

`throw()` was used to indicate that no-exception will be thrown but is deprecated since C++11 and was replaced by `noexcept`section 3.

3. Noexcept

Definition 4.5 Noexcept: Specifies that the (lambda) function, method does not throw any exception.

```
rType myFun(args) noexcept
{
  ...
}
```

Why should we use this?

Provides an optimization opportunity for the compiler but we better make sure that the function does not throw any exceptions.

Attention: If the function throws an exception nevertheless the program just crashes without any indication why.

4. Standard Error Output

Definition 4.6 cerr: Is the standard error output and should be used instead of `cout` in order to output error messages.

```
std::cerr << error_message << "\n";
```

5. Exception Handling Classes

Even empty exception handling classes may be useful if the have a good name indicating the error. They are even better if the provide certain methods to query details about the error. See section 6.

5.1. Std Exception Handling #include⟨exception⟩

Definition 4.7 Standard Base Exception Class: *#include <exception>* defines the standard base class interface exception:

```
class exception {
public:
  exception () noexcept;
  exception (const exception&) noexcept;
  exception& operator= (const exception&) noexcept;
  virtual ~exception();
  virtual const char* what() const noexcept;
}
```

all exceptions derived from `exception` can be catched using:

```
catch(exception& e )
```

`what()`: returns the error message (if available) of the error.

Style it can be useful to write your own exception classes that implement the exception interface. In order to do that we need to overwrite the virtual method `what`.

5.2. Std Exceptions #include⟨stdexception⟩

Definition 4.8 Standard Exceptions: *#include <stdexception>* defines a set of standard errors that are all derived from??. All errors take an optional string as argument to their constructor that should describe the error:

```
throw std::error_type("error meassgae");
```

1. **Logical Errors**: `std::logic_errors`:
   - `std::domain_error`: value outside the domain of the variable
   - `std::invalid_argument`: argument is invalid
   - `std::length_error`: size too big
   - `std::out_of_range`: argument has invalid value
2. **Runtime Errors**: `std::runtime_errors`:
   - `std::range_error`: an invalid value occurred as part of a calculation
   - `std::overflow_error`: a value go too large
   - `std::underflow_error`: a value got too small

see section 6

6. Examples

Example 4.1 Empty Exception Handling Classes:

```
class NoMOreData(){};
int main(){
  try{
    cin >> data;
    if(data==0) throw 1;
    if(data < 0) throw NoMoreData();

  } catch(int intError){
    ...
  } catch(NoMOreData& e){
    ...
  }
}
```

Example 4.2 Standard Exceptions:

```
int main {
  try {
    std::cout << integrate(sin,0,10,1000);
  } catch(std::range_error& e){
    stdd::cerrr << "Range error: " << e.what() << "\n";
  } catch(std::exception& e){       try something more generl
    stdd::cerrr << "Error: " << e.what() << "\n";
  } catch(...){
    stdd::cerrr << "A fatal Error occurred.\n";
  }
}
```

Templates

Templates enable us to do:
- Generic programming
- Template meta programming

types of template programming includes explicit/partial specialization, default template arguments, template non-type arguments and template template arguments.

Definition 4.9 **Generic Programming**: Style of computer programming in which algorithms are written in terms of "types to be specified later" that are then instantiated when needed.

Definition 4.10 **Template Meta Programming (TMP)**: Refers to the use of the C++ template system to perform computations already at compile time within the source code. Hence slower compile time but extremely fast and efficient code.

Substitution Errors Are not necessarily flagged as error

Standard Template Library (STD)

The C++ library includes the same definitions as the C language library organized in the same structure of header files, with the following differences:

1. Each header file in C++ has the same name as the C language version but
   - with a "c" prefix
   - and no ".h" extension

   For example, the C++ equivalent for the C language header file <stdlib.h> is <cstdlib>.
2. Every element of the C++ libraries is defined within the std namespace and not globally.

C-include: `#include<library.h>`
C++-include: `#include<clibrary>`

Timer

10.1. #include⟨chrono⟩

In order to obtain the time passed between two time points, in seconds use:

```
const auto t1 = std::chrono::steady_clock::now();
// ...
const auto t2 = std::chrono::steady_clock::now();
std::chrono::duration<double> diff = t2-t1;
```

or in order to get a different unit use

```
const double diff =
↪  std::chrono::duration_cast<unit>(t2-t1).count();
```

Where unit corresponds to:

- std::chrono::nanoseconds
- std::chrono::microseconds
- std::chrono::milliseconds
- std::chrono::seconds
- std::chrono::minutes
- std::chrono::hours

Input and Output

C++ includes two input/output libraries:

- The standard set of C Input/Output functions
- A modern, stream-based object-oriented I/O library

Note

The C/C++ language did not build the input/output facilities into the language. That is, there exist no specific keywords like read or write.
Instead, it left the I/O to the compiler as external library functions.

12. Stream Based I/O #include⟨ios⟩

Definition 4.1 The Stream Object stream: Is the basic data type in order to represent I/O-operations in C++. A stream is basically a sequence of bytes flowing in and out of a program Thus streams act as intermediaries between the program and an actual IO devices s.a. files, consoles, disks, networks, other programs,... The basics steps to perform input and output in C++ consist of:

1. Constructing a stream object
2. Connect the stream object to an I/O device
3. Perform I/O operations on the stream via the streams public methods/operators
4. Disconnect the stream object from the I/O device and free stream object.

Definition 4.2 std::ios_base: Is the abstract base class of all stream classes.

12.1. #include⟨iostream⟩

Defines standard input/output stream objects

- cin: standard input stream object
- cout: standard output stream object
- cerr: standard output stream object for errors
- clog: standard output stream object for logging

make this a subsubsubsection

Setting the output precision

```
std::cout.precision(17);
```

12.2. #include⟨fstream⟩

Defines file types for manipulating files

- ofstream: used to create or write to files
- ifstream: used to read from files
- fstream: able to do both of the above

Checking for existence

```
std::ifstream fh(filename);
fh.good();
```

opening files

```
stream_type fh;
fh.open(file_path, mode)
// do something
fh.cloes();
```

modes

- ios::app: append to the file
- ios::ate: open the file and move the read/write pointer to the end of the file
- ios::in: open a file for reading
- ios::out: open a file for writing
- ios::trunc: if the file already exists, delete its contents

Note

All this modes may be combined using the | symbol

12.3. Reading and Writing

As fstream objects are streams we may use the same stream operators we use for cin/cout.

Writing to a file

```
fh << value;
```

12.4. #include⟨iomanip⟩

Defines functions in order to manipulate input/output

- setfill: set the streams fill character
- setw: set the streams fill width
- setprec: set the streams output precision for floating-point values

change this better

Setting the output precision

```
std::cout.precision(17);
ostream << std::setprecision(17) << ...
```

13. C-Style I/O #include⟨stdio.h⟩

13.1. fprintf

Definition 4.3: Writes C-strings to the specified output stream:

```
int fprintf(FILE *stream, const char *format, spec_arg1,...)
```

- stream: Pointer to a FILE object that identifies an output stream
- format: is a C-string that contains the text to be written to the stream.
  It can optionally contain embedded format specifiers that are replaced by the values specified in subsequent additional arguments and formatted as requested.
- spec_arg: depending on the format string, the function expects a sequence of additional arguments equaling the number of values specified in the format specifiers.
  Each argument specifies the value to be used to replace a format specifier in the format string (or a pointer to a storage location, for n).

13.1.1. Format Specifiers

Definition 4.4: Specifies format, type and other attributes of a supplied specifier argument and has the following form:

```
%[flags][width][.precision][length]specifier
```

specifiers

specifierspecifies type and interpretation of its argument:

Specifier	Directive	Example
i(d)	signed decimal integer	392
u	unsigned decimal integer	7326
f(F)	decimal float (uppercase)*	392.65
e(E)	Scientific notat. (uppercase)*	3.9265e+2
g(G)	Use shortest f/e (F/E)	392.65
c	Character	c
s	String of chars	sample
p	pointer address	b8000000
n	nothing specified by int	
%	writes % sign	%

Note*

Uppercase corresponds to things like nan/NAN and e/E.

add rest: http://www.cplusplus.com/reference/cstdio/fprintf/

13.2. printf

Definition 4.5 printf: Writes C-strings to the standard output stream:

```
int fprintf(const char *format, spec_arg1,...)
```

Note

Its basically a shorthand for:

```
int fprintf(stdout, const char *format, spec_arg1,...)
```

Type Conversions

14.1. #include⟨cstdlib⟩

Declares a set of general purpose functions for:

- Converting strings
  - std::atoI()
  - std::atof()
  - std::atol()
  - std::atoI()
- Mathematical Tools:
  - std::abs(): returns abs of integer and long integer values

Math

cmath

- std::cbrt: Calcualte the cubic root $\sqrt[3]{\cdot}$

15.1. #include⟨cmath⟩

Declares a set of functions to compute common mathematical operations and transformations

Others

- abs(): returns absolute value of given type.

Algorithms & Data Structures

2. Generic Container

2.1. C-Style Array
2.2. #include⟨array⟩
2.3. #include⟨vector⟩
2.4. #include⟨map⟩

Definition 5.1 Map:

define new code box for this

3. Generic Algorithms #include⟨algorithm⟩

Implement a big number of useful algorithms.
Rely only on the existence of iterators ⇒ do not depend on a specific container type.

Note: C-style Arrays

Can use generic algorithms by using pointers i.e.

```
algorithm(arr, arr+N);
```

3.1. for_each

Applies a function my_function onto each element of a given container and returns the return value of the function call for the last element.

```
for_each(it first, it last, my_function);
```

Create sequences of $2^1, 2^2, \ldots, 2^5$ Listening 5.1

```
vector<int> exp{1, 2, 3, 4, 5};
int base = 2;
std::for_each(exp.begin(), exp.end(), [base](int &val) { val
↪  = pow(base, exp); });
```

add somewhere in algorithms section

Title Listening 5.1

```
void get_metrics(vector<unsigned long long> cycles, Metrics
↪  &metrics) {
  double sum = std::accumulate(std::begin(cycles),
  ↪  std::end(cycles), 0.0);
  double mean = sum / cycles.size();

  double accum = 0.0;
  std::for_each(std::begin(cycles), std::end(cycles),
  ↪  [&](const double val) {
    accum += (val - mean) * (val - mean);
                                                    });

  double  std = std::sqrt(accum / (cycles.size() - 1));
  metrics.mean = mean;
  metrics.std = std;
}
```

C++11

4. Constexpr

5. Special Pointers

5.1. Nullpointer

In c++ NULL is just a define for zero: `#define NULL 0`.
Assume: now want to pass a NULL pointer to a function for example for debuging or if don't know yet which concrete pointer to use.
Problem: if we have an overloded function, the function now no longer knows what NULL is.

```
void f(int v) std::cout << "non-pointer overload";
void f(int* v) std::cout << "pointer overload";

int main(){ f(NULL); }
```

f(NULL) will print "non-pointer overload" eventought we would have expected/wanted the "pointer overload"-version.
Solution: f(nullptr) will print "pointer overload" as expected.

Add different ways of passing functions to functions: https://vittoriomeo.info/index/blog/passing_functions_to_functions.html

6. Function Arithmetics

6.1. Lambd Functions

Are closeures or anonymous functions that are used where you used to (before C++11) create Functors (simple small function objects) that are not meant to be reused.

```
[auto functionHandle =
↪  ][captures](parameters)[->retrurnType]{body}
```

captures:

- []: No external reference.
- [=]: Capture all values from the enclosing scope by value.
- [&]: ——"——by reference.
- [this]: Capture all data members of the enclosing calss.
- [var1, &var2, ...]: Specification for single values.

6.2. std::function

Is a templated object that is used to store and call any callable type, such as functions, objects, lambdas and the result of std::bind

```
#include <functional>
std::function< returnType(argType) > f;
```

6.3. std::bind

std::bind is a template function that that binds a set of arguments to a function and returns a std::function object.

```
#include <functional>
[(std::function <returnType(argTyppe)>|auto) f = ]
↪  std::bind(callableObj, arg1, ...)
```

Listening 5.1 : *Example* *cpp*

```
using namespace std;
void execute(
  const vector<function<void() >>& fs){
  for (auto& f : fs) f();
}

vector<function<void ()>> vec;
function <void ()> f
   = bind(callable, args);
vec.push_back(f);
```

Assume we define a function that takes an vector of void functions as parameter. If we now want to pass a non-void callable to the vector, we can use bind to define a new void function.

Placeholders

std::bind allows us also to use placeholders, which are simply numbers (indicating the arg position in the new function) followed by an underscore.

```
std::bind(callable, 2_, 1_) // arg1 ↔ arg2 reversed
```

6.4. std::future

7. Rvalue Reference

Rvalues vs. Lvazlues Listening 5.2

```
int a = 5;  // a is a lvalue
int& b = a; // b is a (lvalue) reference
int&& c;    // c is a rvalue reference
```

Overloading for r-and l-value references

Given: overloaded function printInt:

```
void printInt(int& i) std::cout << "lv-ref" << i;
void printInt(int&& i) std::cout << "rv-ref" << i;

printInt(a); //Calls printInt(int& i)
printInt(6); //Calls printInt(int&& i)
```

Attention

If we would also define a function void printInt(int a) we would get an compiler error.
This is because for:

1. printInt(a) the compilder does not know if to call the reference or the non-reference version.
2. printInt(6) the compilder does not know if to call the non-reference or the rvalue-reference version.

⇒ can only define two of the three versions.

There are two main applications of rvalue-references:

1. Move Semantics: to optimize data transferring between the objects, in situations when objects have large contents allocated on the heap.
2. Perfect Forwarding: to optimize data forwarding to other functions using universal references.

7.1. std::move

Given Listening 5.3

```
class MyVector{
  double* arr_; //big array
  size_t size;
public:
MyVector(const MyVector& rhs){ // expens. Copy Constr.
  size = rhs.size;
  arr_ = new double[size];
  for(int i=0; i<size; ++i) arr_[i] = rhs.arr_[i];
  }
MyVector(MyVector&& rhs){ // cheap Move Constr.
  size = rhs.size;
  arr_ = rhs.arr_; /* doesn't do any copying
  $\Rightarrow$ need to set the rhs.size to nullptr so that
↪  arr_ doesn't get destroyed when destructor of rhs is
↪  called.*/
  rhs.size = nullptr;
  }
~MyVector() {delete [] arr_; }
}
```

Passing MyVector to functions Listening 5.4

```
main(){
  MyVector reusable = createMyVector();

  foo(reusable); //Deep copy using copy constructor
  foo(createMyVector()) //Cheap copy using move const.
}
```

How can we make use of this?

If we want to write a generic efficient program: we need to distinguish between copying (lvalues) and moving objects (rvalues):

```
foo(MyVector v);
foo_by_refernce(MyVector& v);
```

Problem we do not want to create lots of different versions of foo.
Solution: c++11 offers the std::move function that will move an lvalue with the move constructor.

```
main(){
  MyVector reusable = createMyVector();

  foo_by_reference(reusable);//Cheap copy with move const.
  foo(reusable);             //Most expensive
  foo(std::move(reusable));  // reusable is destroyed here
  // ⇒ reusable.arr_ = nullptr;
}
```

Notes

1. With move we do no longer need to define foo_by_reference.
2. Attention: after std::move we can no longer use reusable.

Conclusion

1. Move semantic avoids costly and unnecassary deep copying and is used by all stl containters.
2. Move Constructors/Assignment Operators are particularly powerfull where passing by value **and** passing by reference are used.

7.2. std::forward

Given a functions that forwards its arguments to another function, perfect forwarding ensures that the argument passed to the second function as if the first function doesn't exist.

```
template<class T>
void wrapper(T&& arg)
{
    // arg is always lvalue
    foo(std::forward<T>(arg)); // Forward as lvalue or as
    ↪  rvalue, depending on T
}
```

Why is this useful: it avoids excessive copying, and avoids the template author having to write multiple overloads for lvalue and rvalue references.

Implementation of std::forward

```
template<class T>
  T&& forward(typename remove_reference<T>::type& arg)
    return static_cast<T&&>(arg);
```

Law 5.1 Reference Collapsing: C++11 defines reference collapsing rules for type deduction in order for the compiler to chose the right type:

Compiler Code	collapsing ⟶	Type
T& &	⟶	T&
T& &&	⟶	T&
T&& &	⟶	T&
T&& &&	⟶	T&&

Definition 5.2 Universal References T&& ref: T&& ref is a universal reference and not only a rvalue-refrence ⟺ :

1. T is a templated type.
2. Type dedeuction (reference collapsing happens to T).

This give functions the power to take on any value (not only type) lvalue, rvalue, const, non-const,...

Listening 5.2 : *Example* *cpp*

```
void fu(X&& t);
void fu(X& t);

template<typename Arg>
void relay(Arg&& arg)
// Universal Reference
{
    fu(std::forward<Arg>(arg));
}

int main(){
  MyVector v;
  relay(v); // lvalue ref (1)
  relay(MyVector()); // rvalue ref (2)
}
```

(1): T is deduced to be an lvalue reference, std::forward<T> just returns its argument and does nothing.
(2): std::forward makes sure that the argument is forwarded as an rvalue reference and thus that fu(X&& t) is called.

Where do we need this?

Essential for libraries such:

1. std::thread
2. std::function which pass arguments to another (user-supplied function).

Random Numbers #include⟨random⟩

Real random numbers are extremely hard obtain: cosmic radiation, quantum random numbers,...
Idea: generate hughe sets off *pseudo random number* algorithmitically.

1. Uniform Random Numbers

1.1. Linear Congruential Generators (LCG)

Definition 6.1 Linear Congruential Generators(LCG):
Use a linear transformation to get the next/a new random number X_{n+1}:

$$X_{n+1} = (aX_n + c) \mod m \qquad X_0 : \text{(Seed)} \tag{6.1}$$

Corollary 6.1 Quality Of RNG:

1.2. Lagged Fibonacci Generators (LFG)

2. Seeding

3. Non-Uniform Random Numbers