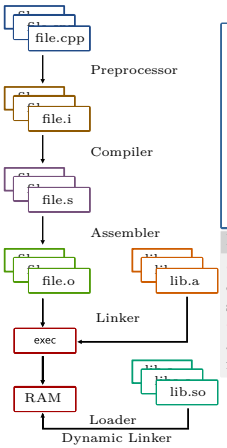
Compiling a C++ Program

Definition 1.1 Source Code/Files `file.c/cpp`: Is the highest level, human readable and writeable form of a computer program that can be translated into machine language with the help of a compiler[def. 1.2].

Definition 1.2 Compiler: Is a computer program or set of programs, that may include a preprocessor??, an assembler[def. 3.1] and or linker[def. 4.1]. 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[def. 1.3] or directly into machine language/binary code.



Definition 1.3 Assembly Language (asm) `file.asm/s`: 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`.

The Preprocessor `*.i`

Definition 2.1 Preprocessor Directives `#`: Are special commands that start with a hashtag `#` and are intended for the preprocessor[def. 2.2].

Definition 2.2 The Preprocessor `CXX -E`: The preprocessor is a program that is a part of the compiler. It performs preliminary operations on `.c` and `.cpp` source files using preprocessor directives[def. 2.2] (except for predefined macro expansion[def. 2.4]) before they are passed to the assembler??. It can be used to conditionally compile code, insert files, specify compile-time-error messages and apply machine specific rules to sections of the code.

```
CXX -E file01.cpp [file02.cpp ...]
```

Preprocessor Command Line Options

2.1 `-Ipath`: appends(≠overwrite) `path` to the include search paths for header files[def. 2.5].

Notes

- Can be used to overwrite a system header file, substituting your own version.
- The preprocessor search for directories in the order the appear on the command line.

2.2 `-Dmacroname[=value]`: defines the macrodefinition 2.2 `macroname` and assigns it the `value` or 1 if no value is specified.

Macros

Definition 2.3 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 2.1 define: lets us define a macro:

```
#define alias macro
```

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

Definition 2.2 undefine: allows to undefine a macro:

```
#undefine alias
```

Note

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

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

Predefined Macros `__Macro__`

Definition 2.4 Predefined Macros : Are predefined macros[def. 2.3] that start and end with two underscores and are expanded by the preprocessor:

```
__Macro__
```

Property 2.1 `__FILE__`: Expands to the current filename.

Property 2.2 `__TIME__`: Expands to the current system time.

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 examples 18.3 and 18.4

Definition 2.3 `#ifdef`: allows to check if a *single* macros is defined:

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

Definition 2.4 `#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
```

Header File Includes

Definition 2.5 Header Files `.h/.hpp`: Header files are simply `c/cpp` source files that (should only) contain:

- Declarations[def. 5.5]
- Inline Functions??
- Templates??
- Preprocessor Macros
- Constants shared by files
- Shared typedefs

Compilation The content of a header file should compile correctly by itself

2.1. Including Header Files

Definition 2.6 `#include`: instructs the preprocessor[def. 2.2] to pastes the entire content of another file into the current file, at the position where we define that detective.

User Includes

Definition 2.5 Quoted Inlcude Files: Includes header files[def. 2.5] by searching for include files ≈in the following order:

1. from the local directory of the include file
2. directories added using the preprocessor option `-I`[opt. 2.1]
3. in `/usr/local/include` – user defined header files
4. in `/usr/include` – system managed (i.e. pacman) header files

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

Querying the include search paths

```
CXX -xc -E -v - < /dev/null
```

System Includes

Definition 2.6 Including Standard Files: The preprocessor searches for header files[def. 2.5] in an implementation defined manner in some predefined directories.

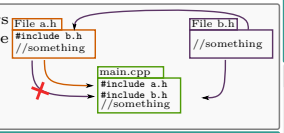
1. directories added using the preprocessor option `-I`[opt. 2.1]
2. in `/usr/local/include`
3. in `/usr/include`

Those libraries can then simply be included by:

```
#include <lib>
```

Include Guards

Help to avoid linker errors when including multiple header files into source files.



Definition 2.7 `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.

Separate Compilation Units

Proposition 2.1

Splitting code up into multiple files: We can separate source code into multiple files by including header files:

```
functions.hpp
function declarations
```

```
functions.cpp
#include "functions.hpp"
function definitions
```

```
#include "functions.hpp"

int main(){
  do_something
}
```

Pros

- Separate compilation of translation units reduces recompilation time. If we change one source file we only have to recompile that source file.
- Searching for certain function, classes or errors in a single huge source files is difficult.
- Teams can work more easily on different parts of a project.

Header File Changes:
If we change one header file, we need to recompile all files that include that header file.

How to choose header files

1. Include standard libraries only into those file, that really need it and not into the header.
2. Only include those headers that you really need.
3. Try to avoid unnecessary dependencies. That is touching one headerfile cause the recompilation of every source file, whether we use this particular change (or even worse header file) at all.
4. From 2 follows avoid including other header file by using forward declarations.
5. Don't force together things that will be used or maintained separately, and don't separate things that will always be used and maintained together.
6. For large project use one header per class to separate translation units, that is if we change one class we don't have to re-compile also all other classes.

Asserts and Errors

2.2. Errors

Definition 2.8 `#error message`: Allows to issue error message at compile time i.e. if certain conditions are not met.

2.3. Asserts `#include <cassert>`

Definition 2.9 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")
```

2.3.1. 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!

The Assembler `*.asm/*.s`

Definition 3.1 Assembler: Assembly language gets converted into object[def. 3.2] files by a program called assembler.

Definition 3.2 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 [def. 4.1].

The Linker `ld`

1. Linkage

Definition 4.1 Linker `ld`**:**
The Linker *ld* is a computer program that takes one or many object files `*.o`[def. 3.2] and combines them into one single executable file, a library, or another object file[def. 3.2]:

```
ld file01.o [file02.o ...]
```

Naming History

`ld` is an abbreviation for *load/loader* even though it is not a loader. This is due to the fact that in the 1970s under UNIX the linker as we know today was actually denoted as loader.

Note: System Libraries

Are usually linked by default.

Static libraries
todo

Linker Command Line Options

4.1 `-L/--library-path`*searchdir*: Specifies the path where `ld` searches for libraries by appending it to *library path*. All `-L` options apply to all `-l` options.

4.2 `-l/---library`*library*: Specifies a library *library* located within the *library path* and to be linked by `ld`. `ld` will first look for a library called

1. *library*
2. dynamic libraries called `lib`*library*`.so`
3. static libraries called `lib`*library*`.a`

Compiling

1. The Translation Unit

Definition 5.1 Translation Unit:
Is the basic unit of compilation in c++. I consists of the contents a single source file `file.c/cpp`, plus the contents of any header files included directly or indirectly by it, minus the lines ignored using conditional pre-processing.
A single *translation unit* can be compiled into an object file `file.o`, a library, or executable program.

1.1. 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*, then 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++[def. 5.20].

Note

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

1.2. 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.

`-Wa,`*opt1[,...]*: Passes a comma seperated list of options to the assembler.

`-Wp,`*opt1[,...]*: Passes a comma seperated list of options to the preprocessor.

`-Wl,`*opt1[,...]*: Passes a comma seperated list of options to the linker *ld* for linking.

1.3. 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[def. 1.3] `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 files[def. 3.2] `file.o`.

1.3.1. GNU `gcc/g++`
1.3.2. LLVM `clang`
1.3.3. Intel MKL `icc`
1.3.4. Microsoft Visual C++ (MSVC) `icc`

Decelerations

Deceleration Paradigm

Definition 5.2 Deceleration Paradigm: Defines the computation logic without defining its detailed control flow.

Definition 5.3 Names/ Identifiers: Variables, objects, namespaces, functions, constants and other entities can be named using *identifiers*.

Definition 5.4 Qualified Names:

Definition 5.5 Deceleration:
A deceleration *declares* a name[def. 5.3] to the compiler and describes its type.
It tells the compiler that a program element/identifier exist and serves as a reference for the *linnker*. The compiler reads files sequentially, hence before we use an identifier[def. 5.3] we have at least to declare it in order for the compiler to know it exists.

Note

A identifier can be declared as often as we want.

Types Of Declarations

Variable Declarations

Definition 5.6 Variable Deceleration:

```
type name;
```

Function Deceleration/Prototype

Definition 5.7 Function Declarations:

```
return_type name(arg_type_list);
```

Class Deceleration

Definition 5.8 Class Declarations:

```
class MyClass;
```

Why would it make sense to only declare classes?

In order to avoid including unnecessary header files.

Forward Declarations

Definition 5.9 [examples 18.5 and 18.6]
Forward Declarations: A forward deceleration is a deceleration of an identifier in order to call the corresponding entity before it has even been defined:

1. Declaration of an entity
2. Referenceing that entity
3. Actual definition of that entity

Definitions

Definition 5.10 Definition: Is an actual implementation of an identifier and allocates storage for it. It is what is needed by the linker in order to link identifiers to the actual implementation.

Corollary 5.1 Variable Declaration's inside functinos:
A variable deceleration inside a function:

```
int main(){ type var;}
```

automatically sets aside memory for that variable and thus is automatically also a *definition*.

Notes

- A definition is also always an deceleration.

Attention:
If we declare a function, do not define it and:

- do not call the function the program will compile fine.
- call the function after it has been declared, the program will compile fine but the linker will complain that it cannot resolve the call.
- call the function before it has been declared, the compiler will already abort.

The One Definition Rule

Definition 5.11 One Definition Rule(s):

1. **In a translation unit**[def. 5.1] each *variable*, `function`, `class type`, `enumeration type`, or `template` can have no more than one definition[def. 5.10].
2. **In the entire program**, an object with *external linkage*[def. 5.18], a *non-inline* function, a class, an enumeration or template cannot have more than one definition.
3. An *inline* function?? hast to be *defined*[def. 5.10] in every translation unit that calls the function and all those function definitions have to be exactly the same.
4. Defining variables in different block with the same name does not violate the one definition rule ; that variable may have the same name but they are distinct.

Inline Functions
are best defined in a header file s.t. only need to include the header files into a file that needs the function.

The Scope

Definition 5.12 Scope: Is a region in which an identifier is visible to other parts of the program.

Definition 5.13 Local Scope:

- Are functions, variables,... that are declared inside a function or control block.
- They are unknown to other functions or control statements outside of their own scope.
- Local variables are linked/prefered before global variables.

Attnetion: Local variables are deleted/deconstructed after their scope is lost.

Definition 5.14 [??]
Global Scope:

- Global identifiers are defined outside of all the functions of a translation unit[def. 5.1], usually at the top of the file.
- They hold their value throughout the entire life of the program.
- A global variable can be used by any function/control statement inside a translation unit.

Linkage

Definition 5.15 Linkage:
Is the way that identifiers in a translation unit are linked to their definitions during the link process.
It is a property that expresses where in a program code an entity represented by an identifier can be reached. An identifier *has linkage* if you can use the identifier to access something inside the programa that is outside of the scope in which the identifier has been declared.
The *type* of linkage for each identifier of a translation unit is determined after the contents of any header files have been inserted into the `.cpp` that is the basis of the translation unit.

Type	Linkage
`global variables`	external
`functions`	external
`global static variables`	internal
`const global variables`	internal
`local variables`	no-linkage

5.1. Internal Linkage

Definition 5.16 Internal Linkage: The entity that the identifier represents can be accessed from anywhere within the same translation unit.

5.1.1. Static and Linkage `static`

Definition 5.17 Static Variables/Functions:
A variable or function declared at global or namespace scope, inside a translation unit[def. 5.1] has internal *internal linkage*. Meaning that you can only use it in the translation unit where it is defined.

```
static Type myIdentiier;
```

5.2. External Linkage

Definition 5.18 External Linkage:
An identifier with external linkage cannot only be accessed inside its own but also in another translation units of the program.

5.2.1. Extern `extern`

Definition 5.19 Extern:
If we want to use global variables with extern linkage in another translation unit we must *forward-declare*[def. 5.9] it by using `extern` keyword.

```
extern Type identifier;
extern const Type identifier;
```

Constant Variables Rather define `const` variables inside a header file instead of using extern.

C++ Concepts

Definition 5.20 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.

6. Programming Paradigms

Definition 5.21 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 5.22 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 5.23 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 5.24 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

Concepts [part of C++20]

Definition 5.25 Concepts:
Concepts are a set of *compile-time* requirements, consisting of *valid expressions*, *associated types*, *invariants* and *complexity guards*. Concepts are completely specified by:

1. Invariants[def. 5.28]
   - Preconditions[def. 5.29]
   - Postconditions[def. 5.30]
2. Complexity Guards
3. Expression Semantics
4. Valid Expressions
5. Associated Types

Definition 5.26 Model: A type that satisfies the requirements of a concept is called a model of that concept[def. 5.25].

Definition 5.27 Refinement: is a concept[def. 5.25] that extends the requirements of another concept.

Definition 5.28 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 5.29 Pre-Conditions: A condition that needs to be satisfied prior to the execution of a function.

Definition 5.30 Post-Conditions: A condition that is ensured to be held after the execution of a function.

Definition 5.31 Complexity Guards: Specifies resource consumption e.g. execution time, memory, #function calls.

Definition 5.32 Expression Semantics: Meaning of the expressions i.e. Assignment

Definition 5.33 Valid Expressions: C++ expressions that are required to compile susccessfully i.e. assignment `x=y;`

Definition 5.34 Associated Types: Name of auxillary types associated with the concept.

Contracts

Definition 5.35 Contracts:
Contracts are a set of *runtime* requirements specified by a set of preconditions, postconditions and invariants associated to a function.

Libraries

1. Statically Linked Libaries/Archieves *.lib/a

1.1. Classical Archives *.a/*.ar/*.lib

Definition 6.1 Archives: Is simply a collection of multiple files with some meta-data composed into a single file, for easier portability and storage. There exist different archive formats that support different features.

Definition 6.2 The Archiver ar: The archiver is a Unix utility that maintains groups of files as a single archive file[def. 6.1]. Today, `ar` is generally used only to create and update static library `.o`,`.obj`.
The file format is not standardized and several variants exit.

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

`-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.

Notes

- A good combination to use is `ruc`.
- In the Linux Standard Base (LSB), `ar` has been deprecated and is expected to disappear in a future release of that Standard.
- Windows extension `*.lib`
- Linux extension `*.a`

Definition 6.3 ranlib: ranlib generates and adds an index for an archive[def. 6.1]. The index lists each symbol defined by a member of an archive that is a relocatable object file. An index speeds up linking to the archive and allows routines in the library to call each other without regard to their placement in the archive.

1.2. Creating Static Libraries

Definition 6.4 Static Library/Archives:
Are a set of object files[def. 3.2] packed into an archive[def. 6.1] that can be linked with an executable or another library.

Definition 6.5 Creating Static Libraries/Archives:

1. Creating an achieve from a set of object files `*.o`:
   ```
   ar [options] liblibname.a file1.o file2.o ...
   ```
2. Add index/table of contents (not needed on all platforms)
   ```
   ranlib liblibname.a
   ```

1.3. Linking Static Libraries

Definition 6.6 Linking Static Libraries:

```
CXX [... [-static] main.cpp libname.a
```

6.1 `-Lpath`: Specify the path from which to include the library

When linking a static library we do not need to inlcude its header file inside our file. However we need to forward declare the functions of the library that we want to use in order for the compiler to compiler our pgoram. However this is error prone and not good practive. => best include the header file of the library. It is possible. You have to declare the funtions that are defined in the .lib manually before you can use them. It is errorprone and is not advised. Headers are not stored in libraries. Headers are stored separately from libraries. Libraries contain object files; headers are not object files.

Attention:

2. Dynamically Linked Libaries *.dll/*.so

Definition 6.7 Dynamic Libraries:
The dynamic linker looks for `.so` files in the following order:

1. in the `LD_LIBRARY_PATH` environment variable (`DYLD_LIBRARY_PATH` on OSX).
2. directories listed in the executable's `rpath??`;
3. directories on the system search path:
   - `/etc/ld.so.conf`
   - `/lib`
   - `/usr/lib`

Definition 6.8 SO/Version Numbers `libname.so.version`**:**
Different version of dynamic libraries are denoted by so-numbers append to the library name.

Corollary 6.1 `ldd`
Querying Dynamic Library Dependencies : We can use the `ld` *dependencies* program in order to query the satisfied and unsatisfied dependencies of an executable:

```
ldd /usr/bin/executable
```

Attention: `ldd` expects the full path to the binary.

3. Header Only Libaries *.h

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

Definition 7.2 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*

Basics

Definition 8.1 [examples 18.1 and 18.2]
Aggregates **C++11 standard:**
An aggregate is an array or a class that has:
- no base classes.
- no user-*provided* constructors.
- for non-static data members:
  - no brace initializers.
  - no equal initializers.
  - no private or protected members.
- no virtual functions

Definition 8.2 Plain Old Data Structure:
Is an aggregate?? class that:
- contains only POD types as members
- has no user defined

todo

Note
In the new standard the concept of POD is split into trivial and non-trivial classes.

Memory Characteristics of Expressions in C++

r-,l- and x-values describe the memory characteristics of different expressions and are important for memory management and function resolution.

1.1. L-values

Definition 8.3 L-values:
L-values are all expressions that refer to a memory location with a persistent address that can be used to access its value. This includes:
- References
- Variables
- array elements
- Dereferenced pointers

Note
Rvalues can be assigned to l-values.

1.2. R-values

Definition 8.4 R-valus:
Are temporary expressions that do not have a identifiable memory address s.a.:
- Literals `"Hallo World!"` or `42`.
- The result of arithmetic or logical expressions (`58+x`).
- Temporary objects whose lifetime is ending `get_value()`

rvalues are stored in CPU registers or on the stack during the evaluation of and expression and are deleted afterwards.

Sidenote lifetime
return value optimization (RVO) or *named return value optimization* (NRVO) is a technique used by the compiler which eliminates the need to copy the return value of a function or to create a temporary object. The compiler can detect when a temporary object is not needed and can reuse the memory that would have been used for the temporary object for other purposes.

1.3. X-values

Definition 8.5 X-Value [C++11]:
X-values are r-values that are to be moved s.t. the lifetime of the r-value is increased to the lifetime of the target object, that the r-value is moved to:

```
    lvalue    rvalue
int   x   =   42  ;
int && rr  = std::move(x);
     rval ref     x-value
```

Pointers:
Pointers are neither lvalues nor rvalues. The are a type of object in C++ to store memory addresses of another object. As such pointers can in theory either point to an lvalue or rvalue (also not meaningful).

Types

Type	Meaning	Size in B
`void`	Incomplete Type	0
`char`	Character	1
`wchar_t`	Wide Char	2
`bool`	Boolean	1
`int`	Integer	2 or 4
`float`	Floatin Point Value	4
`double`	Double Float	8

2.1. Fixed Width Integers [C++11]

Definition 8.6 `#include <cstdint>`
Fixed Width Integers:
Are platform independent integers that guarantee a fixed size:
`B` : 8 16 32 64

Type	At least
`[u]intB_t`	exact no-padding width
`[u]int_fastB_t`	fastest signed integer type
`[u]int_leastB_t`	smallest signed integer type
`intmax_t`	maximum-width signed integer type
`intptr_t`	signed integer type tha can hold ptr to void

Explanation 8.1 (Why non-fixed size integers). *When computers were slow and performance was of the utmost concern. C opted to intentionally leave the size of an integer open so that the compiler implementers could pick a size for int that performs best on the target computer architecture.*

Use fixed width integer if you make assumptions about the width.

Type Modifiers

Definition 8.7 Type Modifiers: Are keywords that can be used to modify simle types:

Type	At least
`short`	2
`long`	4
`long long`	8
`unsigned`	2x (but only positive values)

Note
C++ does not define the actual size of these types (but it does guarantee minimum sizes). On modern architectures, floating point representation almost always follows IEEE 754 binary format.

2.2.1. Constant

Definition 8.8 Constant `const: const`
is a keyword that can make certain properties constant. It applies always to the thing left to it, unless there is nothing left then it applies to the right.

Variables

Definition 8.9 Constant Variables: Are variables that have to be initialized when they are defined and cannot change their value once they have been defined:

```
(const Type|Type const ) var=val;
```

- Use uppercase letters for constants.
- Define shared constants in a shared header file.

Type Casting

Definition 8.10 Type Casting: Is the process of converting one data type into another. There exist two different types of convserions:
(1) Implicit Casting
(2) Explicit Casting

Definition 8.11 Implicit Casting: Is the automatic conversion of types by the compiler. For example when assigning a different type to a variable.

Definition 8.12 Explicit Casting: Is the conversion to types explicitly by the prgorammer.

2.3.1. Static Casts

Definition 8.13 Static Cast: Is the standard conversion that is also used under the hood for implicit conversions of built-in types. The compiler will attempt to convert types even if their is a loss of precision.

```
type var = static_cast<type>(var_other_type);
```

Note pointers
Static cast can also convert related pointers.

Should be used to reveres implicit conversions.

Attention:
- Static casts are type-safe that is we can only down cast but not upcast.
- It does not perform any runtime checks.

2.3.2. Constant Casts

Definition 8.14 Constant Cast:
Changes the *const-ness* of an expression:

```
var = const_cast<type>(const_var);
```

- Can be used to pass a constant variable or pointer to a function that does not accept constant arguments.
- Avoid using this cast rather rethink the structure of your code.

2.3.3. Reinterpret Casts

Definition 8.15 Reinterpret Cast: Is the most powerful cast that does not considere type-saftieness. It can convert:
- from any-built in type other
- from any pointer type to any other pointer type

Note
Can not change the const-ness or volatility of a type.

Should only be used when absolutely necessary.

2.3.4. Dynamic Casts

Definition 8.16 Dynamic Casting: Is a cast at runtime that is mainly used for downcasting polymorphic objects. As such the base class needs to contain at least one virtual function.

```
new_type *ptr = dynamic_cast<new_type>(ptr_orig);
```

Note
- A dynamic cast can also be used for up casting to base class.
- A dynamic cast performs runtime checking, as such its slower than a static cast.

2.3.5. Regular Casts

Definition 8.17 Regular Cast:
Is the most powerful cast that is a combination of `const_cast`, `static_cast` and `reinterpret_cast`:

```
new_type *ptr = reinterpret_cast<new_type>(ptr_orig);
```

Type Information

Definition 8.18 `#include<numeric_limits>`
Numeric Limits:
Provides type information functions:

```
static constexpr return_t numeric_limits<T>::static_fu();
```

Methods

Methods:

Method	Meaning
`epsilon()`	Machine Epsilon
`min()/max()`	Max/min positive normalized value
`is_integer()`	self explainatory
`is_signed()`	self explainatory

Storage class specifiers
- 2.3.6. auto
- 2.3.7. register
- 2.3.8. static
- 2.3.9. extern
- 2.3.10. mutable

Keywords

Volatile

References

Definition 8.19 References &:
A reference is an alias for an already existing identifier:

```
Type &name = entity
```

- A reference is accessed the same way a normal identifier is accessed.
- A reference always refers to the object it was initialized with.
- There exit no *null-references*.

Pros
- Safer than pointers

Cons
- Cannot change address

Implementation
There is no language standard that requires an particular implementation as long as the behavior is compliant. It can be implemented to hold a machine address of an object, and may have less performance overhead than a pointer but is often implemented as a pointer.

Note sizeof
`sizeof` applied to a reference returns the size of the underlying data type and not the size of the reference itself.

4.1. Rvalue References

Definition 8.20 Rvalue References &&:
Rvalue references are references that can bind to rvalues:

```
void foo(int&& x) {
    // do someting
}
auto get_rvalue(){
  return 42;
}
foo(get_rvalue());
```

The advantage of rvalue references is that they avoid copying of temporary objects through call-by-value[def. 8.37]. The lifetime of an r-value reference is the same as the lifetime of the object it references.

Sidenote
An r-value reference is automatically created for temporary objects, and its lifetime lasts until the end of the full expression in which the temporary object was created.

4.2. Constant References

Definition 8.21 Constant References:
Makes the *value* of a refernce constant i.e. we cannot modify it anymore:

```
(const Type|Type const ) &ref=val;
```

Pointers

Definition 8.22 Pointers: Is a variable that:
(1) Stores the address of `&` of an entity.
(2) Allows us to access `*` the corresponding entity.

```
Type *name;                                  Decleration
ptr [= &entity ];                    Address [assignment];
*ptr [= value ];                       Value [assignment];
```

Definition 8.23 Address Operator `&entity`:
Allows us to access the actual address of the memory location where a given entity is stored.

Definition 8.24 Dereference Operator *:
Returns the actual value of memory address a pointer is pointing to.

5.1. Constant Pointers
5.1.1. Constant Address Pointers

Definition 8.25
Pointers with Constant Address:
Are pointers whose address cannot be modified once it has been set:

```
Type *const ptr = &entity;
```

Explanation 8.2. *Constant address pointers are in fact really similar to references.*

5.1.2. Constant Values Pointers

Definition 8.26
Constant Pointers:
Are pointers where we cannot modify the value of the entity where the pointer is pointing to:

```
(const Type|Type const ) *ptr = &entity;
```

5.2. Void Pointer

Definition 8.27 Void Pointer:
A void pointer is a pointer that has no associated data type as such it:
- Can hold addresses of any type
- It can be typcasted to any type
- It cannot be dereferenced
- Pointer arithmetic is not possible

```
void *name;
```

Pros
- Allows functions to allocate memory for any data type; `new` and `malloc` return `void` pointers.
- Allows to write functions that can work with different pointer types.

Cons
- We lose all type information ⇒ can lead to alignment problems.

C++ and typecasting:
In C++ void pointers must be explicitly typecast to a sought after type.

C++ The usage of `void` pointers is usually disencouraged.

5.3. Nullpointer
5.3.1. NULL

Definition 8.28 NULL:
`NULL` is a MACRO that is an implementation defined "null pointer", that indicates that a pointer is not pointing anywhere:

```
#define NULL void*                    C (not allowed in C++)
#define NULL 0                                  before C++11
#define NULL nullptr                             since C++11
```

```
Type ptr = NULL;
```

Cons
- Type decution may be unclear as `NULL` is a an zero int in C++:

```
void f(double *ptr);                          Our ptr function
void f(int v);
f(NULL);                                 Calls the non-ptr function
```

- `NULL` is implementation dependent.
- Pointers may get converted to type `int`.

5.3.2. Nullptr [C++11]

Definition 8.29 Nullpointer `nullptr`:
Is a special type to indicate that a pointer is not pointing to anything and automatically deduces a null pointer of the correct type depending on the type it is assigned to:

```
Type *ptr { [nullptr] };
```

5.4. Smart Pointers #include⟨memory⟩ [C++11]

Definition 8.30 [examples 18.36 and 18.37]
Unique Pointer:
1. Do automatically deallocate the memory that they are pointing to, once they go out of scope.
2. A resource can only have one *unique pointer* pointing to it.

```
std::unique_ptr<Type> p(new Type);                                  or
std::unique_ptr<Type> p = std::make_unique<Type>(size);
```

Definition 8.31 Shared Pointer:
1. Allow in comparison to *unique_ptr*[def. 8.30] multiple references to a single resource by using an internal counter.
2. Automatically deallocate the memory of a resource once no shared pointer is pointing to that resource anymore.

```
std::shared_ptr<Type> p(new Type);                                  or
std::shared_ptr<Type> p = std::make_shared<Type>(size);
```

Attention: Circular references may still lead to memory leaks as the shared counter will never go to zero ⇒ *weak pointers*.

C-Style Arrays
Loops
For Loops
While Loops
Control Statements
if
switch

Functions

Definition 8.32 Function: Functions are reusable pieces of code that perform computations and may depend on arguments an can return a value:

```
return_t fu_name([Type [arg1] [,Type [arg2],...);  prototype
```

```
return_t fu_name([Type arg1 [, Type arg2,...){
  do_something
  [return entity];                                  definition
}
```

```
[Type var = ] fu_name(Type [arg2],...);          function call
```

Definition 8.33 Function Signature:
The signature of a function is a unique identifier of a function that is given by a combination of:
- The *uniqualified name*[def. 8.48] of a function.
- The `class` or `namespace` scope of that name.
- `const` or `volatile` qualifiers.
- The types of the function arguments.
- The template parameters and arguments, if the function is generated from a template function.

Return type: The return type is only part of the signature the function is generated from a function template.

Definition 8.34 Function Arguments:
Are the real values passed to a function.

Definition 8.35 Function Parameters:
Are the names given to the function arguments.

Definition 8.36 Function Prototype:
A function prototype is a declaration of a function that:
- specifies the function's return type
- its function name
- its parameter types

For readability function prototype arguments should have a name.

Attention: Do not return a reference or pointer to a local variable as they are be deleted after the function call.

Function Arguments

Call by Value (CBV)

Definition 8.37 Call by Value:
Is the default method how arguments are passed to functions. The value of the arguments is copied into the actual parameters of the function.
⇒ we cannot modify the original arguments of the function.

```
return_t fu_name(Type [arg] );                          prototype
```

```
[Type var = ] fu_name(entity);                       function call
```

Call-by-value functions can take l-values and r-values (if the function is not overloaded with and r-value version).

Call by References (CBR)

Definition 8.38 Call by Reference:
Passes arguments as references to functions, as such we are able to modify the original entity:

```
return_t fu_name(Type &[arg] );                         prototype
```

```
[Type var = ] fu_name(entity);                       function call
```

Huge Objects Pass large objects by reference as it avoids copying of those objects and make them `const` if they are not supposed to be modified.

Call by R-value References (CBRR)/[C++11]

Definition 8.39 Call by R-value Reference:
Passes arguments as references to functions, as such we are able to modify the original entity:

```
return_t fu_name(Type &&[arg] );                        prototype
```

```
[Type var = ] fu_name(entity);                       function call
```

L-values and implicit conversion:
L-values are implicitly converted into r-values if the function is not overloaded with a normal call-by-reference[def. 8.38] version.

Pointers

Definition 8.40 Pointer as Arguments:
Allows us to pass entities addresses to function in order to work directly on them:

```
return_t fu_name(Type *[arg] );                         prototype
```

```
[Type var = ] fu_name(address);                      function call
```

Notes
- We can use the address operator[def. 8.23] to get the address of an object.
- We have to dereference[def. 8.24] the entity inside the function, if we want to access its value.
- We may directly modify the entity without the need of returning it.

8.1.1. Functions as Arguments
Function Pointers

Definition 8.41 Function Pointers: Are pointers to functions that can be used as arguments of function s.t. we can passe a function to another functions:

```
return_t (*fu_ptr)([Type arg1 [, Type arg2,...);
```

```
[Type var = ] fu_name(fu_ptr);                       function call
```

Note
The act of calling the function pointer automatically dereferences it.

Function Return Values

Return by Reference

Definition 8.42 Return By Reference [example 18.10]:
Allows us to return a reference from a function:

```
&return_t fu_name(Type [arg] );                         prototype
```

8.2.1. Implicit Function Argument Conversions

Definition 8.43 Implicit Function Conversions: Functions may implicitly convert the type of a given argument if it does not result in a loss of information some of those are:

1. **Integer promotions**:
   Smaller integer types are implicitly converted to larger integer types.
2. **Floating-point promotions**:
   Smaller floating-point types are implicitly converted to larger floating-point types.
3. **Derived-to-base conversions**:
   An object of a derived class type can implicitly be converted to an object of its base class type.
4. **Objects to references**:
   Objects maybe converted to referencs.
5. **Const conversions**:
   An object of a non-const type can be implicitly converted to an object of a const type.
6. **Lvalue-to-rvalue conversions**:
   An lvalue (an object that can be modified) can be implicitly converted to an rvalue (a temporary object).
7. **Array-to-pointer conversions**:
   An array can be implicitly converted to a pointer to its first element.
8. **Function-to-pointer conversions**:
   A function can be implicitly converted to a function pointer.
9. **Null pointer conversions**:
   A null pointer constant (such as nullptr or 0) can be implicitly converted to a pointer or pointer-to-member type.
10. **const to non-const**:
    A constant value can be converted to a non-constant call-by-value function call if overloaded const function is given.
11. **Reference to no-reference**:
    A referencecan be converted to a l-value copy call-by-value function call.

References and const: A reference argument cannot be converted into a constant reference. This is because a reference is essentially an alias for an object, and its type must match the type of the object it refers to. A normal object can be implicitly converted to a const reference, but a reference cannot be implicitly converted to a const reference, because a reference is simply an alias for an object, not a distinct object. The type of the reference must match the type of the object it refers to, and cannot be modified once it has been declared.

Static Variables

Definition 8.44 [example 18.9]
Static Variables inside Functions:
Static variables inside functions *are initialized only once* (even for repeated function calls), and hold their value even after the function terminates:

```
return_t fu_name([Type arg1 [, Type arg2,...]){
  static Type var [ = val];
  do_something
  [return entity];                                  definition
}
```

Callback Functions

Function Overloading

Definition 8.45 Function Overloading:
Allows to have a function with the same name but a different signature that is:

- different number or types of arguments
- constness of the arguments
- Reference, pointer or pass-by-value arguments
- Default arguments (implicitly defines functions where you do not need to supply an argument).

Different Return Types:
The return type may be different only if the arguments are different.

Note

- Under the hood C++ defines different function names by appending the types to the function name in order to obey the one definition rule[def. 5.11].
- for templated functions the signature must included the return type, as it is possible and allowed to have tamplated functions which only differ in the return type.

Inline Function `inline`

Namespaces `namespace`

Definition 8.46 Namespace:
Is a named space in which we can define named entities s.a. variables, functions, classes,... in order to distinguish them from entities with the same name.
Definition:

```
namespace ns_name{
  ...                                               definitions
};
```

Usage: there are three ways of using a namespace:

1. Using the *scope resolution* operator:

   ```
   ns_name::entity [something];
   ```

2. The using deceleration:

   ```
   using ns_name::entity;
   entity [something];                          use entity directly
   ```

3. The *using directive* includes everything from a namespace:

   ```
   using ns_name;
   use whatever
   ```

Definition 8.47 Namespace Aliases:
If we want to use the 1 – *scope resolution operator* or 2 – *using deceleration* for a long namespace name we may define a shortcut for it:

```
namespace long_namespace_name ns_shortcut;
```

- Prefer version 1 or 2 as the using directive may lead to conflicts with other namespace.
- Namsepaces are great for bigger projects.

Definition 8.48 Unqualified Names: Is a name that does not appear to the right of a scope resolution operator ::.

Nested Namespaces

Corollary 8.1 Nested Namespaces: Namespaces may be deeply nested:

```
namespace NS1{
  namespace ns2{
    ...                                             definitions
  };
};
```

Anonymous/Unamed Namespaces

Shortcuts

Typedef `typedef`

Definition 8.49 Typedef: Is a shortcut for a complex type:

```
typedef complext_type my_shortcut;                  definition
```

```
my_shortcut var;                                    usage
```

Complete Types:
typedefs may only be used for *complete types*??

Using `using`

Definition 8.50 Using: Is a shortcut for a complex type:

```
using my_shortcut = complext_type;                  definition
```

```
my_shortcut var;                                    usage
```

Pros

- can be used for incomplete types and thus template aliasing (C++11).

9.1.1. Template Aliasing [C++11]

Corollary 8.2 [example 18.8]
Template Aliasing: `using` allows us to define type for *incomplete types*??:

```
template <class T1 [, classT2 [, ...]]>
using complex_template_type = my_shortcut;
```

Complex Types

Data Structures `struct`

Definition 8.51 Structs:
A struct is a special data type that allows us to define our own data types, that combine data types of different type:

Enums

Definition 8.52 [example 18.11]
Enum `enum`:
An enum is a user defined data type, that consists of named integral constants:

```
enum Ename {
  enumerator1 [ = int] [, enumerator2 [ = int] [,...]] }
};
```

```
Ename var [ = (int/enumerator)];
```

11. Initialization

Definition 8.53 Initialization**:** Is the process of setting the initial value of a variable.

Definition 8.54 Initializer**:** Specifies the initial value of a variable

11.1. Kind of Initialization

11.1.1. Zero Initialization

Definition 8.55 Zero Initialization**:**
Zero initialization is performed for:

- For every named variable with *static* or *thread-local* storage duration, that is not subject to constant initialization.
- As part of value-initialization[def. 8.57] for a sequence for non-class types and for members of value-initialized class types that have no constructors, including value initialization of elements of aggregates for which no initializers are provided.
- When an array of any character type is initialized with a string literal that is too short, the remainder of the array is zero-initialized.

The effects of zero initialization are:

1. If `T` is a scalar type, the object's initial value is the integral constant zero explicitly converted to `T`.
2. If T is array type, each element is zero-initialized.
3. If T is reference type, nothing is done.
4. If T is an non-union class type, all base classes and non-static data members are zero-initialized, and all padding is initialized to zero bits. The constructors, if any, are ignored.
5. If T is a union type, the first non-static named data member is zero-initialized and all padding is initialized to zero bits.

11.1.2. Default Initialization

Definition 8.56 [example 18.13]
Default Initialization**:**
Is an initiaialization[def. 8.53] performed with an object without any initializer:

- `T object;`
- `new T;`
- When a base class or a non-static data member is not mentioned in a constructor initializer list and that constructor is called.

The effects of *value initiaialization* are:

1. For non-POD objects (until C++11) class types the default constructor is called to provide an inital value of the new object.
2. if `T` is an array type, every element of the array is default-initialized.
3. Otherwise, for POD object **no initialization is performed**. The objects with automatic storage duration (and their subobjects) contain indeterminate values!

11.1.3. Value Initialization

Definition 8.57 [example 18.14]
Value Initialization**:**
Is an initiaialization[def. 8.53] performed with an empty initializer of the form:

- `T()`
- `new T ()`
- `T()`

C++11:

- `T object{};`
- `T {};`
- `new T{};`
- `Class::Class(...) : member() {...}`

The effects of *value initiaialization* are:

1. If `T` is a class type with no default constructor or with a user-provided or deleted default constructor, the object is *default-initialized*??.
2. If `T` is a class type with a default constructor that is neither user-provided nor deleted, the object is zero-initialized **and** the semantic constraints for default-initialization are checked.
3. If `T` is an array type, each element of the array is value-initialized.
4. Otherwise the object is *zero-initialized*??.

11.1.4. Direct Initialization

11.1.5. Copy Initialization

11.1.6. Brace Initialization

Narrowing

11.1.7. Member Initializer Lits

11.1.8. Uniform Initialization

Classes

Definition 9.1 Classes:
Classes are boiler plates for reusable data types with additional functionality:

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

Style
- Use capital letters for class names.
- use underscore suffixes for member variables `var_`
- `*.hpp`:
  - Member Functions & Variables Declarations
  - Non-const static variables initializations outside of the class
- `*.cpp`:
  - Member variables with constructor
  - Member function definitions

Definition 9.1 Member:
Are variables, functions, objects defined with our class.

Definition 9.2 Member Access:
1. By reference `.`:
```
MyClass obj{};
[T var = ] obj.memberVar [ = val ];
[T var = ] obj.memberFu([args]);
```
2. By pointer `->`:
```
MyClass *objp = &obj;
[T var = ] optr->memberVar [ = val ];
[T var = ] optr->memberFu([args]);
```
3. Static class members `::` :
```
[T var = ] MyClass::staticVar [ = val ];
[T var = ] MyClass::staticFu([args]);
```

Explanation 9.1 (`->` is just a shortcut).
`ptr->member;` ≙ `(*ptr).member;`

Definition 9.3 `this`: Is a pointer to the object itself, and can thus only be used inside the class.

Don't use `this->memberVar;` to distinguish member variables from parameters; rather user underscore suffixes for to distinguish member variables i.e. `memberVar_`.

Auto Generated Member Functions

C++03:
1. The Default Constructordefinition 9.2
2. The Copy Assignment Operator
3. The Destructor

C++11:
4. The Move Constructor
5. The Move Assignment Operator

Creating Objects

Automatic Allocation

Definition 9.4 Object Creation:

```
MyClass obj;                          default initialization
MyClass obj(args);                    Value/Direct initialization
MyClass obj{[args]};                  Value/Direct Initialization
```

Prefer to create default objects by means of an empty set of braces {} (value initialization).

Dynamic Allocation

Definition 9.5 Dynamic Object Creation:

```
MyClass obj
```

Access Modifiers

1.1. Public

Definition 9.6 `public`:
All members[def. 9.1] are accessible for all and everyone.

Encapsulation
- Try to avoid public members as much as possible, as they break encapsulation; if something goes wrong it is much harder to track down the problem.
- Within the code define `public` before `protected` before `private` members.

1.2. Protected

Definition 9.7 `protected`:
Protected members are only accessible:
1. From within the class itself
2. With classes *derived*?? from that class.

1.3. Private

Definition 9.8 `private`:
Private members of a class are only accessible with the code of the class itself.

The Constructor

Definition 9.9 Constructor:
A constructor is a `public` member function that has the same name as the class/struct and is used to initialize the class members.

The rule of 5
When you require a destructor, we should also explicitly take care of the copy constructor, the copy assignment operator, the move constructor and the move assignment operator.

Explanation 9.2. *This is beacuse destuctors are usually called for dynamic memory allocations.*

2. Default Constructor

Definition 9.2 Default Constructor a:
A default constructor is a constructor without any arguments or only arguments with a default value:

```
class MyClass{
public:
  MyClass([ arg1=val])[:member_initializer_lists]{
    default initialization                    ctor body
  }
}
```

```
MyClass ob;                          calling the default constructor
```

Note
- Default constructors only allows arguments with an default value that can be called with empty arguments.
- The order in which members are initialized is defined by the order in which they are declared within the class.

Attention:
- If non-default constructors are declared, the compiler will not provide a `default` constructor.
- While we can use `MyClass obj1(args);` to create objects using the non-default constructors. The call `MyClass obj2();` is a definition of a function and not the call of the default constructor!
- `MyClass obj2();` corresponds to *default-initialization* and not *value-initialization*. Thus POD types will not contain a meaningful value.

Corollary 9.1 Default Construction of Members:
- Any member of type `class`, `struct` that is not initialized in the initializer 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

Member Initializer Lists

Initializing elements inside a ctor is not always possible. This is because the objects of a class are already constructed before the constructor body but
- non-static `const` variables
- references

must be assigned during construction, as they cannot be modified after initialization.
⇒ need to use member initializer lists, to set their value during construction.

Definition 9.10 Member Initializer Lists:
Member initializer list specifies the initializers for direct and virtual base classes and non-static data members of the class:

```
class MyClass{
public:
    MyClass(arg1[=val][, ...]) :  mem_{arg1} [, ...]  {
        default initialization                    ctor body
    }
    T mem_;
    ...
}
```

Note
We can use either () or {} initializers.

Core Guideline C.47 Define and initialise member variables in the order of member declaration.

Core Guideline C.49 Prefer initialization to assignment in constructors.

Why use Member Initializer Lists

Pros
- for performance reasons: we do not first default construct and object and change its value using an assignment seeexample 18.16
- We cannot not initialize:
  - non-static `const` members
  - references

  inside the ctor body seeexample 18.17
- Initialization of subobjects which only have parameterized constructors i.e. no-default ctor.
- Initialization of baseclasses which only have parameterized constructors i.e. no-default ctor.
- Initialization of baseclasses with a non-default ctor.

2.1. In Class Initialization [C++11]

Definition 9.11 [example 18.18]
In Class Initialization: In-class initializers are used if the data member is not explicitly listed in the member initializer list.

Note
The compiler is smart enough to know which value to use and it avoids extra assignments inside the ctor body.

Pros
- avoids code duplication i.e. writing the same initializer list that varies only by a variable for ctor that vary only by one variable.
- Makes it explicit that the same value is expected to be used in all constructors.
- Using in-class member initializers lets the compiler generate the function for you.

Core Guidlines C.45 Don't define a default constructor that only initializes data members; use in-class member initializers instead.

Core Guideline C.48 Prefer in-class initializers to member initializers in constructors for constant initializers.

2.2. Auto Generated Default Constructor

Definition 9.12
Auto Generated Default Constructor: For a `class`/`struct` without any user defined constructor the compiler will generate an auto-generated `default` constructor. The auto generated constructor initializes all members

2.3. Explicit Default Constructors

Definition 9.13 `default` [C++11]
Explicit Default Constructor: If we provide any user defined constructor, the compiler will no-longer provide an auto-generated *default* constructor s.t. we have to provide our own if we want to be able to default construct our class. However we can explicitly ask the compiler to do this for us by using the `default` keyword:

```
MyClass{
public:
    MyClass(args){
      something                              non-default ctor
    }
    MyClass() = default;          ask to provide default ctor
}
```

3. Copy Constructor

Definition 9.14 The Copy Constructor **a:**
Is an auto generated constructor (if not specified explicitly) that allows us to copy objects:

```
MyClass{
public:
    ...
    MyClass(MyClass [const|volatile] &obj);    auto generated
    ...
}
```

```
MyClass obj1{};
MyClass obj2(obj1);                            Copy Constructor
```

When do we have to write our own?

The default copy ctor does memeberwise copying i.e. pointers are copied and not the object that they are pointing too. Thus for a deep copy we need to copy the object the point to by ourself.

Conversion Constructor
Move Constructor
General Constructor
Member Initializer Lists

5. Uniform Initialization

6. Aggregate Initialization

Member Functions/Methods

Constant
Constant Methods

Definition 9.3 Constant Methods:
are functions that are *not allowed* to change any *class members*.

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

Mutable

Definition 9.15 [example 18.15]
mutable: The mutable qualifier allows modification of member variables in constant member functions and even in constant objects:

```
mutable Type var;
```

Explanation 9.3. *This is helpful if we want for example to count the number of function calls for a constant member fucntion of an object.*

Constant Method Arguments

Definition 9.4 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 [,...]);                     declaration
```

Friend Functions

Definition 9.16 Friend Functions: A friend function is a non-member function or member function of another class that is granted access to the private and protected members of a class:

```
class A {
    private:
     int x;
    public:
     friend void print(A &obj);
};
void print(A &obj) {
    cout << obj.x;
}
```

7.5. Setters & Getters

Definition 9.17 Setter & Getters:
For non-public members we still might need a way to access those members. A convenient way is to define functions for setting and getting those members:

```
void var_(Type val){ var_ = val; };            setter
```

```
void var_ const (){ return var_; };            setter
```

Style

- If we define private members using an underscore suffix we can conveniently use the member name for the setter and getter functions.
- If we do nothing but rereading & writing of variables we may avoid using setters.
- Avoid public members as well as setters & getters as much as possible as they break encapsulation.

Pros

- When we realize that we need to do more then just set and/or get a variable, we do not have to change the code for every location where we set and/or get the variable.
- We may perform validation on our variables.
- We can hide the internal representation of the variables i.e. getting several fields by one method and vice versa.
- If we have data that needs to be read but under no circumstances modified we can only implement a getter.

Creating & Acessing Objects

Static

8.2. Static Variables

Definition 9.5 Static Member/Variable:
Are variables that exist once *per class* (and not per object):

```
class Foo                                      decleration
{                                              .hpp
  private:
     static T var;
};
```

1. The initialization of static members is not allowed inside the header file (`.hpp`) as it would violate the one definition rule[def. 5.11]. 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.

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

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

Explanation 9.4. *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.*

Note

- Static and thread storage duration objects without value initialize are zero initialized: 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.

8.3. Static Functions

Definition 9.18 Static Functions:
Are functions that exist once *per class* (and not per object): Static functions may only access static members.

```
class Foo                                      deceleration
{                                              .hpp
    static T fu([args]);
};
```

```
T Foo::fu([args]) = {                          definition
  definition                                   .cpp
}
```

```
[T var = ] MyClass::fu([args]);                call
```

Notes

- `this` keyword is not available in a static member function.

8.4. Static Objects

Definition 9.19 Static Objects: Static objects are allocated at program startup in static storage and exist for the entire life of the program.

```
static MyClass obj;                            default initialization
```

```
static MyClass obj(args);                      Value/Direct initialization
```

```
static MyClass obj{[args]};                    Value/Direct Initialization
```

Note

The constructor will be called at the end of the lifetime of the program.

Operators

Definition 9.20 Ad-hoc Polymorphism:
Operator overloading is known as ad-hoc polymorphism in which operators are treated as polymorphic functions, that behave differently depending on the type.

Definition 9.21 [example 18.19]
Right Associativity:
States that operators are evaluated from the right to the left.

Definition 9.22 Operator Overloading **Ω:**
Most operators are actually special functions of classes and can be redefined. Operators can be overloaded as member functions or non-member functions:

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

Definition 9.23
Free/Non-member Function Operators:
Are general operator functions:

```
T operatorΩ([args]){
    definition
};
```

Definition 9.24 Member Function Operators:
Reduces the number of arguments by one; the leftmost operand becomes the `*this` object:

```
class MyClass{                                 .hpp
  T operatorΩ([T obj]){
    definition                                 decleration+definition
  }
}
```

or alternatively split it up:

Deceleration

```
class MyClass{
  T operatorΩ([T obj]);
}
```

+

.hpp definition

```
inline T                                       .hpp
MyClass::operatorΩ([T obj]){
    definition
}
```

or alternatively as .cpp definition

```
MyClass::operatorΩ([T obj]){
    definition
}                                              .cpp
```

Explanation 9.5 (Inline).

- *`this` is a pointer to a reference, thus we need to dereference it.*
- *If we define the overloaded operator inside the header file outside of the class and we include the header file into multiple files we need to use the `inline` keyword.*

- Operators should only be overloaded when their use would be natural & unambiguous.
- Operators should be implemented as members if:
  - they require direct access to the non-public members of a class and/or
  - if they change the left hand operator.
- Implement them as non-members/free functions only if it makes sense.

8.5. Assignment Operators *aΩ*: `/= += -= *= &= ^= |= %=`

Definition 9.25 [example 18.21]
Assignment Operators:

1. Can only be implemented as member functions
2. Should always return a `const &`

Explanation 9.6 (
Why to use references). *When considering what we should return from a member function we should look at the expression `a=b=c` and right associativity[def. 9.21]:*

1. *Don't return pointers: as we would have to write `a=*(b=c)`*
2. *Don't return by value as this reduces performance:*

copy `c` → `c'` ⇒ *`b=c'`;* *destruct `c'`*
copy `b` → `b'` ⇒ *`a=b'`;* *destruct `b'`*

Explanation 9.7 (Why to use const). *Here is the part where it gets tricky:*

- *If we want to be able to write expressions like `a=b` and want to make sure that `a` does not change `b` we have to pass `const &`*
- *If we want to be able to chain expression such as `a=b=c` we also have to return constant references `const &`.*

Attention:
Cons

- we are not able to write expresion such as:
  `(a=b).non_const_method();`
  But we can still write:
  `a=b; a.non_const_method();`
- `const &` do not model the standards behavior i.e. we cannot write expression such as:
  `(x=y)=z` as `(x=y)` is already a constant reference.
  But we would anyway never write something like that.

Copy Assignment Operator

Definition 9.26 The Copy Assignment Operator a:
Is an auto generated operator (if not specified explicitly) that allows us to copy assigned objects:

```
X{
public:
    ...
    auto generated
    [const] X& operator=(X [const|volatile] &obj);
    ...
}
```

```
X obj1{};
X obj2 = obj1;                    Copy Assignment Constructor
```

8.6. Symmetric Operators $a\Omega b = b\Omega a$

Definition 9.27 [example 18.20]
Symmetric Operators: Are operators that are commutative $a\Omega b = b\Omega a$. As such they are best implemented as free/non-member operators:

Why do we use non-member function's?

Otherwise we could not use type conversion i.e. `10+obj` would not be possible.

8.7. Unary Operators ++ - + - ! *

Definition 9.28 Unary Operators: Are operators that have only one argument. Sometimes they can be applied from the left or from the right.

1. As member operators:
   Applied from the left Ω`obj;`: `const A &A::operator`Ω`();`
   Applied from the right `obj`Ω`;`: `const A A::operator`Ω`(int);`
2. As free operator:
   Applied from the left Ω`obj;`: `const A &operator`Ω`(A&);`
   Applied from the right `obj`Ω`;`: `const A operator`Ω`(A&, int);`

Explanation 9.8. *`int` is used to indicate if the operator is called from the left or the right.*

8.8. Special Operators

8.8.1. Subscript Operators [] ()

Definition 9.29 Container Type Operators: For container like operators we want to be able to access members by reference or value.

```
class Arraylike{
    public:
    double operator[](size_t i) const{
      return p_[i];
    }
    double& operator[](size_t i) const{
      return p_[i];
    }
    private:
      double* p_;
}
```

Multi dimensional classes: For multidimensional classes use commas `[row,col]` or `(row,col)` instead of double brackets `[row][col]` or `(row)(col)`, as they pose a lot of problems.

8.8.2. Operator Pointers * ->

Definition 9.30 Pointer Like Operators:
For classes that act like pointers:
- Iterators
- Smart Pointers

we want to be able to use pointer syntax:
- `*pObj;`
- `pObj->f();`

```
class P{
  double* operator->(){
    return p_;
  }
  const double* operator->() const {
    return p_;
  }
  double& operator*() {
    return *p_;
  }
  const double& operator*() const {
    return *p_;
  }
private:
  p_;
}
```

8.9. Functors

Definition 9.31 [example 18.22]
Functor:
Are classes that overload the () operator in order to server as function type object.

```
class MyFunc {
  ...
  double operator() double(x) {
    return ...
  }
  ...
}
```

Pros
- can state
- Potentially more efficient than passing a function pointer, as the functor is already known at compile time ⇒ might lead to inlining.

Cons
-

8.10. Stream Operators ≫

Definition 9.32 Stream Operators ≫:
We can overload the input/output stream operators of classes to specify how object's are send to streams:

```
std::stream& operator <<(std::ostream& out, const Point& p)
{
  out << "( " << p.x() << ", " << p.y() << " )";
  return out;
}
```

Notes
- we have to return a reference to the object in order stream multiple objects:
  `cout << pta << ptb << std::endl;`
- `std::cout` is used for writing specifically to the standard output.

Inheritance

Definition 9.33 Runtime Polymorphism: Uses virtual functions:
- function selection at run-time
- one function created for the base class → saves space
- virtual function call needs lookup → slower

Is most useful for application frameworks, user interfaces, big functions.

Definition 9.34 Inheritance: Allows us to write a new class by extending an existing one:

```
MyClass: visibilit_mode BaseClass {
  ...
};
```

Class can be:
1. **Augmented** by using virtual functions.
2. **Overridden** by overwriting all functions.

Note
Classes may be derived from multiple classes.

8.12. Dynamic vs. Static Binding

[example 18.23]

When we have a pointer pointing to an object, the type of the object may either be either of a base class or of a class derived from this base class.

Definition 9.35 Static Type:
Is simply the type of a variable.

Definition 9.36 Dynamic Type:
Is the type of a variable/pointer during runtime.

Definition 9.37 Static Typing/Binding:
The legality of a member function invocation is checked at the earliest possible moment during compile time. The compiler uses the `static type` of the pointer to determine the type of the object.
Non-virtual member functions are resolved statically during compile time. Hence the member function is determined based on the type of the pointer or reference of the object.

Definition 9.38 Dynamic Typing/Binding:
The legality type of the member function invocation is determined at the last possible moment, based on the dynamic type of the object at runtime.
`virtual` member functions are resolved dynamically at ru time. Hence the member function is selected at runtime based on the type of the object not by the type pointer/reference to that object.

Explanation 9.9 (title).

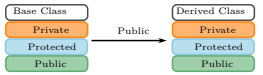
Visibility Mode

Definition 9.39 Visibility Mode:
Specifies how the member variables & functions derived from the base class are accessible.

Definition 9.40 Private visibility (default):
Public and protected members become private as well:

Definition 9.41 Protected Visibility:
Public and protected members become protected:

Definition 9.42 Public Visibility:
Public, protected, private members keep their visibility level.



Virtual Functions

Definition 9.43 Virtual Functions:
We can overwrite functions of the base class dynamically by using the `virtual` keyword:

```
BaseClass {
  virtual fu(args){
    ...
  }
};
```

It is sufficient to only declare the function of the base class to be `virtual`, but for multiple inheritance it is good style to also declare the function in the derived base classes virtual.

Definition 9.44 The vTable:
Dynamic type binding using virtual functions is achieved by an array called vTable. Every class containing one or more virtual functions has exactly one vTable that contains a pointer to each virtual function. Each object of the class obtains a pointer to this vTable. A derived class obtains a copy of this vTable and for each overridden member function the pointer of the vTable gets changed to the member function of the derived class. As such each object of a virtual class occupies more memory than the sum of its element members. Most of the time this additional pointer is neglectable however.

Note
Virtual function calls are slower due to the lookup inside the vTable.

8.13.1. Override [C++11]

Definition 9.45 Override:
Is a keyword to indicate that we override a function of the base class. It lets the compiler check if a `virtual` function with the same name exists inside the base class:

```
MyClass visibilit_mode BaseClass {
  override fu(args){
    ...
  }
};
```

8.13.2. Virtual Destructor

Definition 9.46 Virtual Destruct:
A destructor should be virtual if:
- Someone will derive fro your class, which contains virtual functions.
- Someone will create derived classes from your class on the heap.
- Someone will call delete on base class pointers.

Explanation 9.10. *This is necessary in order to call the destructor of the derived class and not the one of the base class.*

Simply make the destructor `virtual` if there exist any virtual functions.

Interfaces

Definition 9.47 Interface: Describes the behavior or capabilities of a class without committing to a particular implementation of that class.

Note
In C++ interfaces are declared using abstract base classes.

8.13.3. Pure Virtual Functions

Definition 9.48 Pure Virtual Function:
A pure virtual function is a member function that must be overridden in a derived class and is not defined inside the base class:

```
virtual Type fu([args]) = 0;
```

8.13.4. Abstract Class

Definition 9.49 [examples 18.24 and 18.25]
Abstract Class:
An Abstract class is a class containing at least one *pure virtual function*[def. 9.48].

Bitwise Operators

Working with Bits

Definition 9.50 Positional Number System: Is a number system where the value of a number corresponds to a weighted sum of digits. each digit is associated with a weight that is equal to the base/radix b to the power of the position of that number:

$$N = (a_n a_{n-1} \cdots a_1 a_0)_b \qquad a \in \mathbb{N}, \quad 0 \leqslant a \leqslant b-1 \qquad (9.1)$$

$$N = a_n \cdot b^n + a_{n-1} \cdot b^{n-1} + \cdots + a_1 \cdot b^1 + a_0 \cdot b^0$$

Note

The position starts from zero.

Definition 9.51 Base/Radix b**:** Is the number of digits b, starting from zero that are used to represent a single number.

Definition 9.52 Binary Number Format: Is the positional number format[def. 9.50] for the base 2:

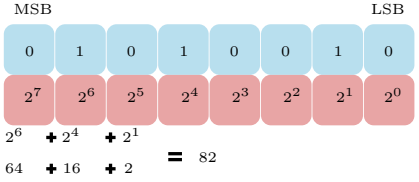


Figure 1: Little Endian Binary Number representation

Definition 9.53 Least Significant Bit (LSB): Is the bit of lowest size in a binary number representation.

Definition 9.54 Most Significant Bit (MSB): Is the bit of largest size in a binary number representation.

Definition 9.55 Big Endian System: LSB is on the right.

Definition 9.56 Little Endian System: LSB is on the left.

Bitwise AND &

Definition 9.57 AND **&:**

Only true if both of the bits are true:

0	0	1	1
0	1	0	1
0	0	0	1

Bitwise OR |

Definition 9.58 Or **|:**

True if one of the bits is true:

0	0	1	1
0	1	0	1
0	1	1	1

Bitwise XOR ^

Definition 9.59 Exclusive OR (XOR) **^ :**

Only true if only one of the bit is true:

0	0	1	1
0	1	0	1
0	1	1	0

Bitwise Complement ~

Definition 9.60 Complement/Not **~:**

Only true if both of the bits are false:

0	0	1	1
0	1	0	1
1	0	0	0

Bitwise Shift Operator «/»

Definition 9.61 Left Shift Operator **«:**

Given a number of n bits it shifts all bits to the left by adding a zero to the right:

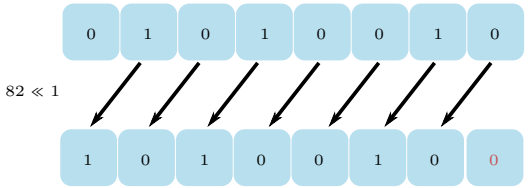


Figure 2: Left shift operator for 8bit number 82

Definition 9.62 Right Shift Operator **»:**

Given a number of n bits it shifts all bits to the right by adding a zero to the left:

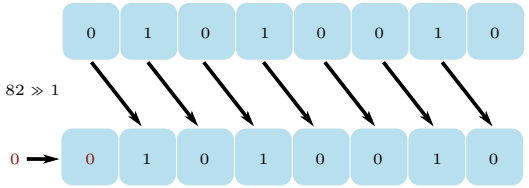


Figure 3: Right shift operator for 8bit number 82

Addition -

Definition 9.63 Addition:

There exist for rules:

0+0	1+0	0+1	1+1 (add to next free)
0	1	1	10

Note

If the both bits are true and the next bit of the resulting number is true as well, they bit gets propagated until it reaches a zero bit.

Subtraction -

Definition 9.64 Subtraction:

There exist for rules:

0-0	1-0	0-1 (borrow from next true)	1-1
0	1	1	0

Note

We cannot subtract 1 from zero thus we need to subtract from the next higher digit.

Working with Binary Numbers

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 10.1 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 10.1 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 10.2 Throw: Allows us to throw/create errors if certain undesirable conditions happen:

```
throw someObj;
```

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

Corollary 10.2 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 10.3 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 10.4 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`[def. 10.5].

3. Noexcept

Definition 10.5 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.

Standard Errors

Std Exception Interface #include⟨exception⟩

Definition 10.6 Standard Exceptions Interface:
Is the standard base exceptions interface class that all other standard exceptions are derived from:

```
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.

Note

Even empty exception handling classes may be useful if the have a good name indicating the error See example 18.27.

Style Own exception classes should implement the exception interface. In order to do that we need to overwrite the virtual method `what`.

Std Exceptions #include⟨stdexception⟩

Definition 10.7 [example 18.28]
Standard Exceptions:
Standard exception's defined in *#include <stdexception>* are 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
3. `std::bad_alloc`
4. `std::bad_cast`
5. `std::bad_typeid`
6. `std::bad_exception`

Standard Error Output #include⟨iostream⟩

Definition 10.8 cerr:
Is the standard error output and should be used instead of `cout` in order to output error messages.

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

Standard Current Exception #include⟨exception⟩

Definition 10.9 `std::current_exception`
Standard Current Exception:
Is a function that during exception handling (in a `catch` clause) gives access to the current exception object by returning an `std::exception_ptr` that holds either a copy or a reference to the current exception object:

```
std::exception_ptr eptr;
try {
  ...
} catch(...) {
  eptr = std::current_exception();
}
handle_expection(eptr);
```

4.2.1. Standard Exception Pointer

Definition 10.10 `std::exception_ptr`
Standard Exception Pointer:
The standard exception pointer `std::exception_ptr` is a nullable pointer that is used to pass standard exceptions around.

4.2.2. Throwing Exception Pointers

Definition 10.11 `std::rethrow_exception`
Standard :
Throws an exception pointed to by a standard exception pointer[def. 10.10].

```
void handle_eptr(std::exception_ptr eptr)
{
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
      ...
    }
}
```

Templates

Definition 10.12 **Compile Time Polymorphism:** Uses templates and:
- Decision which function to use at compile time
- Works for objects having the right members
- A new function created for each used class → needs more space
- Functions are inlined → fast function calls

Useful for small, low level constructs, small fast functions and generic algorithms.

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 10.13 **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 10.14 **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.

Definition 10.15 **Independent Name:**

Definition 10.16 [example 18.26]
Dependent Name:
Is an identifier that depends (directly or indirectly) on a template parameter.

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

8.1. #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.

Stream Based I/O — #include⟨ios⟩

Definition 11.1 Stream: A stream is a sequence of bytes flowing in and out of a program. A streams act as intermediaries between the program and an actual IO devices s.a. files, consoles, disks, networks, other programs,...

Note

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 11.2 std::ios_base:
`std::ios_base` is the abstract base class that all stream objects inherit from.
It hold state and control information of streams and can be used to set basic settings.

Static Methods

`sync_with_stdio(bool = true)`: Controls whether to synchronize/block between C-style `cstio` and C++ standard streams.

Corollary 11.1 Synchronization and Speed:
Synchronization between C–and C++–style I/O may considerably slow-down C++ I/O streams, thus if we can guarantee that C–and C++–style I/O are not mixed we may disable it.

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

#include⟨iostream⟩

Streams

Cin

Definition 11.3 Cin:
Character **in**put stream, is the standard input stream. It is a global variable of the type `std::istream`.

Methods

Definition 11.4 The input operator >>: Cin input operator >> is blocking and reads content from the standard C input stream–`stdin` of a computer. It reads a word and removes any spaces, tabs or line breaks until it can read word of the given data-type:

```
cin >> word1 >> word2 [>> ...];
```

New line characters: The `cin` operator >> reads only the values of the data-type but leaves delimiter characters inside the input stream:

```
// stdin: 5\n{}Hallo World!
int x;
cin >> x;                    \nHallo World! inside the stream
```

Methods:

Method	Meaning
`getline(istr&, &str, del='\n');`	extracts characters from the input stream and appends it to the string object until the delimiting character is encountered.
`ignore(count=1, del='\n')`	reads the next `count` number of characters from stdin or until the delimiter `delim` is reached and discards them.
`clear()`	clears the cin error flag see [example 18.12].

Corollary 11.2 Skipping all possible words in a lines:

```
cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

#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

`stream.setprecision`: Sets the number of total digits of the integral plus decimal part of a number for a given stream:

```
stream.setprecision(digits);
```

Note

This behavior changes if `fixed` or `scientific` notation is used.

Static Methods

`std::setprecision`: Sets the number of total digits of the integral plus decimal part of a number for a given stream on the fly:

```
stream << std::precision(digits);
```

`std::fixed`: Changes the behavior of `setprecision`, to set the number of digits of the decimal part only. This may include trailing zeros.

`std::scientific`: Changes the behavior of `setprecision`, to set the number of digits of the decimal part only and represents a number in scientific notation:

$$m \times 10^n$$

Where $m \in [1, 10)$ and n usually consiting of three digits.

Note

In both the fixed and scientific notations, the integral digits before the decimal point are not relevant for the precision in this case.

Streams

1.1. #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

1.2. 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;
```

C-Style I/O — #include⟨stdio.h⟩

1. Format Specifiers

Definition 12.1: Specifies format, type and other attributes of a supplied specifier argument and has the following form:

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

specifier specifies 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
`zu`	`size_t`	34
`c`	Character	c
`s`	String of chars	sample
`p`	pointer address	b8000000
`n`	nothing specified by int	
`%`	writes % sign	%

Note*: uppercase vs. lowercase

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

Note: i vs. d

Only important for input i.e. `scanf`.
- `%d` scans integer as a signed decimal number.
- `%i` scans input:
  - scans integer as hexadecimal if preceeded by `0x`.
  - scans integer as octal if preceeded by `0`.
  - scans integer as a signed decimal number otherwise.

length modifies the length of the data type:

Length	Variable Type
`h`	short/unsigned short
`l`	long/unsigned int
`l`	wide char/string
`L`	long float

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

Input

2.1. fscanf

Definition 12.2 fscanf:
Reads C-strings to the specified output stream:

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

- `stream`: Pointer to a `FILE` object that identifies an output stream
- `format`: is one or multiple (by spaces seperated) `C-string format specifiers` that are replaced by the values specified in subsequent additional arguments.
- `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).

2.2. scanf

Definition 12.3 scanf:
Reads C-strings from the standard output stream:

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

Note

- Its basically a shorthand for:
  `int fprintf(stdout, const char *format, spec_arg1,...)`
- It is much faster then `std::cin`.

Output

3.1. fprintf

Definition 12.4 fprint:
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).

3.2. printf

Definition 12.5 printf:
Writes C-strings to the standard output stream:

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

Note

Its basically a shorthand for:
`int fprintf(stdout, const char *format, spec_arg1,...)`

3.3. Flushing

Definition 12.1 fflush:
Any buffered unwritten data to the associated output stream are immmediately written from the stream's buffer to the output device:

```
int fflush(FILE *stream);
```

Type Conversions

4.1. #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 $\sqrt[3]{\cdot}$

5.1. #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

Iterrators

Definition 13.1 [examples 18.30 and 18.31]
Iterator [implementation 18.1]:
Iterators are objects that point to the address of an element of a collection. Iterators can be advanced using the increment operator and can be dereferenced to obtain the underlying element.

```
[value =] *it;          reading
*it = value;            writing
```

```
container<type>::iterator it = [pos];
```

Obtaining Iterators: todo

```
auto it1 = container.begin();
auto it2 = container.end();
```

Definition 13.2
End of collection Iterator:
For a collection of n elements the last iterator will point to *past-end-itarator* which can for example be a `nullpointer`.
This itartor is used to indicate if we have reached the end of the collection: `it != end_it`

Why do we need Iterators?

They define a common interface for traversal, access, positional comparison and allow to apply algorithms onto all containers?? that define iterators.

What is an Iterrator

```
container<type> a, b;
size_t size = a.end()-b.begin();
```

1. Ranges

Definition 13.3 Range: A range is the set of two iterators [f, l) that define which on which element of a collection to operator on.

2. Obtaining Iterrators

Obtaining Iterators

Most containers provide methods to access their iterators.

`container.begin()`:
returns the iterator to the first element of the container.

`container.end()`:
returns the iterator to one element *past* the container.

`std::begin(container)`:
returns the iterator to the first element of the container.

`std::end(container)`:
returns the iterator to one element *past* the container.

Generic Container

C-Style Strings `#include<string>`

Definition 14.1 C-Style String `"A C string"`:
Are pointers to the first element of contiguous memory of `Char` that have a implicit sentinel value of `'\0'`.

```
char str[SIZE];          Stack Allocated
char *str = new char[SIZE];   Heap Allocated
```

Strings `#include<string>`

Definition 14.2 String:
Are null-terminated with an invisible `NUL` character, as such the number of charactes as Bytes plus one.

Methods

Methods:

Method	Meaning
`find(seq, size_t pos)`	searches for `seq` starting from `pos` and returns the `pos` of the first match otherwise `string::npos`.
`replace(pos, len, rep)`	deletes `pos` until `pos+len-1` from a string and replaces the deleted part with `rep`.

Operator Methods

Methods:

Method	Meaning
`str1 == str2`	checks if two strings are equal.

Static Methods

Methods:

Method	Meaning
`npos`	represent the end of a string and is defined as `static const size_t npos = -1;` This is the greates representable value of `size_t` and corresponds to 18446744073709551615 on 64bit systems.

Freestanding Methods

Methods:

Method	Meaning
`str1 == str2`	checks if two strings are equal.

Definition 14.3 ASCII (1960):
Uses 7bits to represent 128 Characters, most characters represent normal characters wiht others representing control sequences. ASCII characters where initially stored as 7bits with the 8-th most significant bit set to 0. Modern

dec	oct	hex	ch	dec	oct	hex	ch	dec	oct	hex	ch	dec	oct	hex	ch
0	0	00	NUL (null)	32	40	20	(space)	64	100	40	@	96	140	60	`
1	1	01	SOH (start of header)	33	41	21	!	65	101	41	A	97	141	61	a
2	2	02	STX (start of text)	34	42	22	"	66	102	42	B	98	142	62	b
3	3	03	ETX (end of text)	35	43	23	#	67	103	43	C	99	143	63	c
4	4	04	EOT (end of transmission)	36	44	24	$	68	104	44	D	100	144	64	d
5	5	05	ENQ (enquiry)	37	45	25	%	69	105	45	E	101	145	65	e
6	6	06	ACK (acknowledge)	38	46	26	&	70	106	46	F	102	146	66	f
7	7	07	BEL (bell)	39	47	27	'	71	107	47	G	103	147	67	g
8	10	08	BS (backspace)	40	50	28	(	72	110	48	H	104	150	68	h
9	11	09	HT (horizontal tab)	41	51	29	)	73	111	49	I	105	151	69	i
10	12	0a	LF (line feed - new line)	42	52	2a	*	74	112	4a	J	106	152	6a	j
11	13	0b	VT (vertical tab)	43	53	2b	+	75	113	4b	K	107	153	6b	k
12	14	0c	FF (form feed - new page)	44	54	2c	,	76	114	4c	L	108	154	6c	l
13	15	0d	CR (carriage return)	45	55	2d	-	77	115	4d	M	109	155	6d	m
14	16	0e	SO (shift out)	46	56	2e	.	78	116	4e	N	110	156	6e	n
15	17	0f	SI (shift in)	47	57	2f	/	79	117	4f	O	111	157	6f	o
16	20	10	DLE (data link escape)	48	60	30	0	80	120	50	P	112	160	70	p
17	21	11	DC1 (device control 1)	49	61	31	1	81	121	51	Q	113	161	71	q
18	22	12	DC2 (device control 2)	50	62	32	2	82	122	52	R	114	162	72	r
19	23	13	DC3 (device control 3)	51	63	33	3	83	123	53	S	115	163	73	s
20	24	14	DC4 (device control 4)	52	64	34	4	84	124	54	T	116	164	74	t
21	25	15	NAK (negative acknowledge)	53	65	35	5	85	125	55	U	117	165	75	u
22	26	16	SYN (synchronous idle)	54	66	36	6	86	126	56	V	118	166	76	v
23	27	17	ETB (end of transmission block)	55	67	37	7	87	127	57	W	119	167	77	w
24	30	18	CAN (cancel)	56	70	38	8	88	130	58	X	120	170	78	x
25	31	19	EM (end of medium)	57	71	39	9	89	131	59	Y	121	171	79	y
26	32	1a	SUB (substitute)	58	72	3a	:	90	132	5a	Z	122	172	7a	z
27	33	1b	ESC (escape)	59	73	3b	;	91	133	5b	[	123	173	7b	{
28	34	1c	FS (file separator)	60	74	3c	<	92	134	5c	\	124	174	7c	|
29	35	1d	GS (group separator)	61	75	3d	=	93	135	5d	]	125	175	7d	}
30	36	1e	RS (record separator)	62	76	3e	>	94	136	5e	^	126	176	7e	~
31	37	1f	US (unit separator)	63	77	3f	?	95	137	5f	_	127	177	7f	DEL (delete)

```
int i=4;
char c = 'A'-1+i // -> D
```

Corollary 14.1 Upper vs. Lowercase Letters:
Lowercase letters are always represented as numbers 32 greater than their respective uppercase letters. Thus lower case letters can be switched to uppercase letters and the other way around by setting the single bit corresponding to 32 i.e. the 6-th bit (1 for lowercase, 0 for uppercase).

Corollary 14.2 Extended ASCII: There exist several extensions to ASCII that use the 8-th bit, in order to represent 256 Characters.

Definition 14.4 Unicode: Universal Character Encoding is an encoding that uses 2^{21} characters, which maps the number 0-2^{21} to characters. Unicode is a superset of ASCII, as the numbers 1-127 map to the same characters as in Unicode. Not all numbers are yet assigned. Unicode

Definition 14.5 Unicode Code Points: The Unicode characters are written as hexadecimal number with a U+ prefix:

$$U+0748 \tag{14.1}$$

Why Unicode?

ASCII was defined primarily for the English language but there exist many more languages as well as symbols.

Definition 14.6
Unicode Transformation Format (UTF):
UTF is an encoding system for Unicode. Unicode per-say stores characters in a Unicode format. This format however is not binary as needed by computers.

- **UTF-8** represents a single Unicode characters as either one, two, three or four Bytes. The first Byte corresponds to the ASCII encoding.
- **UTF-16** represents a single Unicode characters as either two or four Bytes i.e. units of 16 bits.
- **UTF-32** is a fixed length encoding of 32 bits.

Note

UTF-8 uses less memory than UTF-16 on the majority of websites. UTF-16 is however more efficient than UTF-8 on some languages if they have characters at the back of the Unicode encoding. UTF-8 and UTF-16 use different algorithms to encode and decode characters and are therefore not compatible.

Contigious Memory Structures

1. C-Style Array

2. Arrays `#include<array>`

Definition 14.7 Array:
Is a container that represents a sequence of fixed size.
Construction:

```
std::array<type, size> a;                  undefined values
std::array<type, size> a = {};             default initialization
std::array<type, size> a = {values};       value initialization
```

Access:

```
a[idx] [ = val];
a.at(idx) [ = val];          Checks idx & returns a reference
```

Operations	Woerst-Case Complexity
Access to arbitrary elements	$\mathcal{O}(1)$

Why use Arrays over C-Style arrays

Arrays do provide consistent copy semantics to be passed to and from functions by value. It is design to have not have any overhead over C-style arrays.

3. Vectors `#include<vector>`

Definition 14.8 Vector:
Is a container that represents a sequence whose length may change dynamically.
Construction:

```
std::vector<type> v;                       zero size vector
std::vector<type> v(size);                 Zero initialized values
std::vector<type> v(size, val); Initialization to same value
```

Access:

```
v[idx] [ = val];
v.at(idx) [ = val];      Checks idx & returns a reference
```

Nested Vectors:

```
typdef std::vector<type> VT1D;
typdef std::vector<std::vector<type>> VT2D;
VT2D(size, VT1D(size, val));
```

Operations	Woerst-Case Complexity
Insert/Add at the front	$\mathcal{O}(n)$
Insert/Add at given idx	$\mathcal{O}(n)$
Append an element	$\mathcal{O}(n)$ ($\mathcal{O}(1)$ amortized)
Del. ele. from the back	$\mathcal{O}(1)$
Access element at given idx	$\mathcal{O}(1)$
Find an element	$\mathcal{O}(n)$

Methods

Methods:

Method	Meaning
`reserve()`	Allows to allocate space beforehand.
`front()`	Accesses the first element.
`back()`	Accesses the last element.
`clear`	Erases all elements from the vector s.t. its size is zero. The capacity per standard is unchanged.
`push_back(val)`	Appends an element to the end of the vector and increments its size.
`pop_back()`	Removes the last element of the vector in decrements its size.
`erase(it+offset)`	erase an element with given index from a vector.
`erase(it+offset, it2+offset2)`	erase a range from a vector.

Why reserve space

Makes `push_back` a woerst-case constant time operation.

Amortized Push Back

Calling `push_back` once is of woerst-case $\mathcal{O}(n)$ but a call of n `push_backs` takes $\mathcal{O}(n)$ time, leaving a single call with amortised complexity $\mathcal{O}(n)$.

4. Deque `#include<deque>`

Definition 14.9 Deque:
Double **E**nded **Que**ue is a vector like container that allows fast insertion at the begin as well as the end.

Operations	Woerst-Case Complexity
Insert/Delete at the front	$\mathcal{O}(1)$
Insert/delete at given idx	$\mathcal{O}(n)$
Insert/Delete at the back	$\mathcal{O}(1)$
Access element at given idx	$\mathcal{O}(1)$
Find an element	$\mathcal{O}(n)$

Methods

Methods:

Method	Meaning
All of `std::vector`	
`push_front(val)`	Prepends an element to the beginning of the vector and increments its size.
`pop_front()`	Removes the first element of the vector and decrements its size.

Lists

Definition 14.10 Lists: Is a container of non-contiguous memory nodes, that are linked by using pointers. There exist two different approaches:

```
struct Node {
  data_t data{};
  Node *next{nullptr};
  Node(data_t data) : data{data} {}
};
```

```
class List{
    void push_front(data_t) {[example 18.34]};
    void push_back(data_t) {[example 18.33]};
    ~List(){[example 18.35]
private:
    Node *head{nullptr};
}
```

Note: linked list without `list` `struct`

It is also possible to let the user create a link list manually without a `list` `struct`, by creating a head node first and by appending to it.
Problem: if we have multiple different objects that point to the head of the linked list and some function or object changes the head of the list, all other data structures will point to a wrong node or even worse to something undefined.

Note: push_back vs. push_front

Push front does not require us to iterate through the whole list i.e. $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$.

Performance Comparison With Vector

From a theoretical standpoint inserting new elements inside the middle of a vector/dequeue should be slow and insertion into a list should be fast. This is due to the fact that the vector memory is contiguous s.t. we need to copy half of the elements. For the list on the other hand we only need to change two pointers. However in reality one needs to consider the advantage of the cache for the contiguous memory of the vector:

- `std::vector` and `std::dequeue` perform always faster than std::list with very small data.
- `std::list` should only be used for random inserts and remove of large elements.

5. Single Linked List #include⟨forward_list⟩

6. Double Linked List #include⟨list⟩

Tree Like Structures

7. Set #include⟨set⟩

Definition 14.11 Set:
The set is a container that contains *unique* elements of a certain type, that are sorted using the < operator.
Sets are usually implemented as *red-black-trees*??.

Construction:

```
std::set<type> s;                          empty set
std::set<type> s {val1, ...};
```

Finding Elements:

```
std::set<type>::iterator it = s.find(val);
if the set is empty: it == s.end()
```

Erasing Elements:

```
std::set<type>::iterator it = s.find(val);
s.erase(it);
```

Operations	Woerst-Case Complexity
Inserting an element	$\mathcal{O}(\log n)$
Finding an iterator	$\mathcal{O}(\log n)$
Deleting an element	$\mathcal{O}(n)$ ($\mathcal{O}(1)$ amortized)

Note

The tree structure needs to be re-balanced, after certain operations.

Methods

Methods:

Method	Meaning
`insert`	Insert an element into the set. Returns a pair, `first` – an iterator pointing to newly inserted element or the one that was already there. `second` is true if a new element was inserted or false if it was already there.
`clear`	Erases all elements from the set s.t. its size is zero.
`lower_bound(val)`	Returns an iterator to the first element that is not smaller than `val` $\mathcal{O}(\log n)$ element≥val
`upper_bound(val)`	Returns an iterator to the first element that is larger than `val` $\mathcal{O}(\log n)$ element>val

Corollary 14.3 Min and Max Element: The minimum and maximum element of a set can simply be found by using iterators:

```
type min_e = *s.begin();
type max_e = *s.rbegin();
```

8. Unordered Set #include⟨unordered_set⟩

Definition 14.12 Unordered Set (Hash Map):
An unordered set is a container that is a set of unordered unique *not sorted* elements. The elements are organized into buckets by a hash of its keys.

Operations	Woerst-Case Complexity
Inserting an element	$\mathcal{O}(n)$ ($\mathcal{O}(1)$ on average)
Finding an iterator	$\mathcal{O}(n)$ ($\mathcal{O}(1)$ on average)
Deleting an element	$\mathcal{O}(n)$ ($\mathcal{O}(1)$ on average)

Attention: Unordered sets are highly dependend on the hash function, storing custom data structures with a bad hash function may lead to the woerst time complexity.

(Unordered) Maps and Dynamic Programming

It may be tempting to use `std::map` or `std::unordered_map` as DP table. But, in particular if the state consists only of integers it is much faster to use a d-dimensional `std::vector` with $\mathcal{O}(1)$ access vs. $\mathcal{O}(\log n)$. However `std::map` can be useful if the DP state consists of serval different data types.

9. Map #include⟨map⟩

Definition 14.13 Map:
Are ordered dictionaries/look-up tables of ordered key-value pairs, where the order is enforced through the < operator over they keys.
Maps are usually implemented as *red-black-trees*??.

```
map<key_t, val_t> m;
m.insert(std::pair<key_t, val_t>(key, value));
m.insert(std::make_pair(key, val));
m.insert({key, val});                                   C++11
```

Construction: Access:

```
m[idx] [ = val]; Ret. ref.to elem.&default-init. if no key
m.at(idx) [ = val];     gives error if key doesn't exist
```

Operations	Woerst-Case Complexity
Inserting an element	$\mathcal{O}(n)$
Access an element by key	$\mathcal{O}(n)$
Find/Remv. an elem. by key	$\mathcal{O}(n)$
Find/Remove an element	$\mathcal{O}(n)$

Note

```
m.insert({key, val});                                   C++11
```
is basically syntactic sugar to:
```
map<key_t, val_t>::value_type(key, val);
```

Methods

Methods:

Method	Meaning
`erase(key)`	erases element with key `key` and returns 1 if the element existed and 0 otherwise.
`emplace(key, key)`	inserts a values with a given key using possible move semantics.

Note

The standard does not require them to be implemented that way.

10. Multimap #include⟨map⟩

Definition 14.14 Multimap:

11. Unordered Map #include⟨unorderd_map⟩

Definition 14.15 Unordered Map: Unordered maps are container that contains key-value pairs with unique key-value pairs but are not sorted. Elements are organized into buckets depending on a hash of the key. They are usually implemented as hash tables.

Attention: Unordered maps are highly dependend on the hash function, storing custom data structures with a bad hash function may lead to the woerst time complexity.

Queue Like Structures

12. Stack #include⟨stack⟩

Definition 14.16 Stack: Stacks are FILO (first-in-last-out) data structures. Stacks are usually implemented using `std::dequeu`[def. 14.9].

```
std::stack<type> s;
s.push(val);                          adds an element to the top
auto top = s.top();                   returns the top element
s.pop();                              deletes the top element
```

Operations	Woerst-Case Complexity
Push/Inserting an element	$\mathcal{O}(1)$
Pop/Delete an element	$\mathcal{O}(1)$
Access the top element	$\mathcal{O}(1)$

Sidenote

Great for depths first search.

13. Queue #include⟨queue⟩

Definition 14.17 Queue: Stacks are FIFO (first-in-first-out) data structures. Stacks are usually implemented using `std::dequeu`[def. 14.9].

```
std::queue<type> q;
q.push(val);                          adds an element to the end
auto front = q.front();               returns the last in q.
auto back = q.back();                 returns first in q.
s.pop();                              deletes the last in q.
```

Operations	Worst-Case Complexity
Push/Inserting an element	$\mathcal{O}(1)$
Pop/Delete an element	$\mathcal{O}(1)$
Access the top element	$\mathcal{O}(1)$

Sidenote

Great for breadth first search.

14. Priority Queues #include⟨queue⟩

Definition 14.18 [example 18.32]
Priority Queues:
Is a queue like structure where the elements are sorted by priority instead of the point of arrival.
By default, the largest elements are at the top (first out). It is usually implemented using vectors. Priority queue are great to be used as min or max heap.

```
std::priority_queue<type> q;
q.push(val);
auto top = q.top();
q.pop();
```

Operations	Woerst-Case Complexity
Push/Insert an element	$\mathcal{O}(n)$ ($\mathcal{O}(\log n)$ amortized)
Access the top element	$\mathcal{O}(1)$
Pop/delete an element	$\mathcal{O}(\log n)$

Algorithms

Utility Library #include⟨utility⟩

1. Swap

Definition 15.1 Swap: Exchanges the values of two variables of the same type:

```
std::swap(var1, var1);
```

Style Improves code redability.

Algorithm Library #include⟨algorithm⟩

2. Max & Min

Definition 15.2 min:
Returns the minimum of two elements:

```
std::min(var1, var1);
```

Definition 15.3 max:
Returns the maximum of two elements:

```
std::max(var1, var1);
```

2.1. Ranges

Definition 15.4 min_element:
Returns the minimum over a range:

```
std::min_element(it1, it2);
```

Definition 15.5 max_element:
Returns the maximum over a range:

```
std::max_element(it1, it2);
```

3. Sorting

Definition 15.6 sort:
Sorts any linearly ordered random acccess container with n elements with worst-case complexity of $\mathcal{O}(n \log n)$:

```
std::sort(it1, it2);
```

Corollary 15.1 Ordering: The ordering is defined by default through the smaller operator <.

Corollary 15.2 Descending Sort: A range can be sorted in descending order by using the `std::greater` function:

```
std::sort(it1, it2, std::greater<type>());
```

4. Shuffeling

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);
```

0.1. for_each

Definition 16.1 [example 18.29]
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);
```

Algorithms on Sequences #include⟨numeric⟩

Accumulate

Definition 16.2 Accumulate:
Accumulates values of a container:

```
accumulate(firstIt, lastIt, init [, op])
```

Allowed Operations: Any binary function or operator that takes two values and returns a value that can be assigned to `T`.

Notes

- The initial values specifies the type i.e. use for zero double *double*.
- `accumulate` performs the accumulation in order.

Attention Type Deduction:
Type deduction is performed on the initial value.

Reduce

Definition 16.3 Reduce **[C++17]:**
Reduce values of a container:

```
reduce(exec_pol [, firstIt, lastIt, init, op)
```

Transform Reduce

Definition 16.4 Transform Reduce **[C++17]:**
Transforms the pairs of two containers and subsequently reduces them:

```
reduce(exec_pol [, firstIt, lastIt, init, op)
```

C++11

1. Constexpr

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

2. Function Arithmetics

2.1. 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.

2.2. 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;
```

2.3. 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 16.1 : *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
```

2.4. std::future

3. Rvalue Reference

Rvalues vs. Lvazlues Listening 16.1

```
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.

3.1. std::move

Given Listening 16.2

```
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 16.3

```
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 unnecessary 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.

3.2. 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 16.1 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 16.5 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 16.2 : *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).

Multithreading #include⟨thread⟩

Multithreading vs Multiprocesses

1. **Multiprocessing**
   - Relative expensive to create/more overhead.
   - Independent resources, state.
   - **Immune to many concurrency issues.**
   - Can run on distributed systems.
2. **Multithreading**
   - `class{}; main()` is the first thread.
   - Cheap to create.
   - Shared resources, state ⇒ interthread-communication is cheaper than interprocess communication.
   - Difficult to use correctly.

Definition 17.1 [example 18.39],[example 18.39]

`std::thread`:

```
template <class Func, class... Args>                Decleration
std::thread::thread(Func&& f, Args&&... args)
```

Threads can be launched by passing a callable object and optionally its arguments to the thread constructor:

```
#include <thread>
std::thread t(callable_object, [args])
```

To wait for a thread to finish we call the blocking join function:

```
t.join();
```

Compilation:

```
[g++/clang++] -std=c++11 -lpthread ... main.cpp -o main
```

Corollary 17.1 [example 18.40]

What kind of objects can we pass to `std::thread`:

`std::thread` is a templates function which allow us to use `std::thread` with any callable and a lot of the standard libraries functions such as `std::bind`, `std::async` `std::call_once`,...

Notes

- Before C++11 POISX Threads had to be used.
- The implementation of std::thread in C++ is based on platform-specific APIs for threading and synchronization, such as pthreads on Unix-based systems, Windows threads on Windows, or Apple Grand Central Dispatch on macOS.

Methods

```
id get_id() const noexcept;:
```

Returns unique ID of thread, important to avoid deadlocks.

```
bool joinable() const;:
```

Checks if thread is still attached to the main thread see section 17.

Passing Parameters

Definition 17.2 `std::ref`:

`std::ref` creates a `std::reference_wrapper` *object*. This object is a *type* that be passed by-value to a function but encapsulates a reference to its underlying object:

```
void add_one(int x) {x += 1; }
int a = 5;
add_one(std::ref(a)); // a is now 6
```

this can be useful for functions that do not take references.

Parameters are always passed to threads by value when creating a thread (even if we pass a reference). If we want to pass a parameter to a thread not as copy we can:

- create a shared memory reference wrapper[def. 17.2]:
  `std::thread t(std::ref(fu) [,args]);`
- Use a pointer `T *arg`
- Move values:
  `std::move(Type& arg)`

Characteristics of Threads

It is not possible to copy/assign from lvalues:

```
std::thread t1, t2; t1=t2; //not possible
```

Note

Non-copy ability is implemented using rvalues and is meanigfull as it does not make any sense to copy a thread.

`thread& operator=(thread&&) noexcept;`

We can change the ownership of a thread:

```
std::thread t2 = std::move(t1);
t2.join(); //instead of t1 now
```

Can copy/assign from rvalues:

```
pool[3] = std::thread(f);
```

Can Swap:

```
t1.swap(t2); // swap(t1,t2);
```

Important library functions #include⟨thread⟩

`static unsigned hardware_concurrency();:`
Gives a good guess on the number of avaiblable cores.

`static void sleep(const system_time& xt);:`
Lets the thread sleep for a certain time.

`std::this_thread::yield();:`
Is a function in the `std::this_thread` namespace that is used to voluntarily give up the CPU and allow other threads to run.

Running Threads in the Background

Definition 17.3 Detach: If we have a function and we do not want to wait at the end of the function for a thread to finish we can detach it and run it as *deamon*:

```
void background();
void someFunction(){
    std::thread t(background);
    ...
    t.detach();
} // no problem here, thread is detached
```

Note

- For a deamon process the c++ runtime library wil be responsible to reclaime the resources of t, after t is finished.
- One can join/detach a thread only once – once detached/joined forever detached/joined.
- If a thread is neither detached nor joined, and the main thread finishes before the thread is done, the thread will simply be destroyed and the program terminates doing nothing.

Attention:
If we want to avoid unplaned deamon processes we need to make sure that in case of an exception (of the main thread) the child threads still join. We can do this either with a try/-catch block or RAII.

Thread Concurrency

Definition 17.4 Resource Acquisition Is Initialization:
Create a wrapper class whos destructor takes care of resource deallocation (release).

Listening 17.1 : *try catch*

```
void fu() std::cout << "";
int main(){
std::thread t(fu);
try{...} catch(...) {
    t.join();
    throw;
  }
  t.join();
}
```

Listening 17.2 : *RAII*

```
/* Use Wrapper, that automatically
↪  calls join() when it goes out of
↪  scope*/
class Wrapper{
  ...
  ~Wrapper(){t_.join}
}
...
```

Definition 17.5 Oversubscription: Using more threads than available cores. Will degrade performace because of unnecessary processor switching.

1. Mutual Exclusion `std::mutex mu;`

Guidlines

1. Use mutex to synchronize data access between threads.
2. Do not use `std::mutex` own `mu.lock()` and `mu.unlock()` method, as in case of an exception `mu` might stay locked forever. Rather use a `std::lock_guard` (see: section 1) that uses RAII:
   - `std::lock_guard< mutexTpye > lock([mu])`
   - `std::unique_lock< mutexTpye > lock([mu])`
3. Make sure that the shared resource is completly under the protection of the lock e.g. do not use global variables e.g. `std::cout` as it can still be accessed at other places. Rather use a own file stream.
4. Design intefaces appropriately.

1.1. `std::lock_guard`

Can only be locked once and gets automatically unlocked when it goes out of scope.

Types of mutexes

1. `mutex`.
2. `recursive_mutex`: allows multiple locking by the same thread.
3. `timed_mutex`: allows time-outs in lock attempts.
4. `recursive_timed_mutex`: both of the above.

Note

For time locks we have to use a timed mutex.

Listening 17.3 : *Examples synchronize I/O* *cpp*

The order is still randomnly **but** each thrad will print a whole line by itslef.

```
std::mutex mu;  // global

struct sync{
  sync( std::ostream& os )
  :os(os), lock(mu) {}

  template <class T>
  std::ostream& operator<<(T
↪   const& x)
    return os << x;
private:
  std::ostream& os;
  std::lock_guard<std::mutex>
↪   lock;
};
```

```
void printer( int n ) {
  for ( int i = 0; i < 100; ++i)
↪  {
    sync(std::cout)
    << "do not garble thread "
    << n<<":"<<i<<"\n";}}
int main(){
  std::vector<std::thread> v;
  for(int n = 1; n < 10; ++n)
  v.push_back(
    std::thread(printer, n)
  );
  for (std::thread& t : v)
    t.join();
}
```

1.2. `std::unique_lock`

Provides more flexibility than `std::lock_guard` **but** is slightly more expensive.

For example it allows unlocking of the mutex if there is still a bunch of other stuff to be done that does not require the lock to be locked:

```
std::unique_lock lock(mu); // locking
// accessing data
lock.unlock(); // other stuff that does not need sync
```

Unique Lock Initalization

```
unique_lock<mutex> l(m [,option]);
```

- `std::defer_lock`: To define a lock (its owner) without locking it yet. Handy if we want to do some stuff first, that does not need locking yet.
- `std::adobpt_lock`: Adopts the lock state ⇒ in case its locked I become the new owner. Important for deadlocks in conjunction with `std::lock` (see section 2).
  `std::adopte_lock` does not lock the mutex object on construction, assuming instead that it is already locked by the current thread.
- `std::try_to_lock`: Tries to lock the lock and returns whether it succeded.
- `abs_time`: Tries to lock the lock with timeout.

Important Functions of unique lock

- `l.owns_lock()` or `if(l)`: Return if the lock is locked.
- `l.try_lock()`: Tries to lock the lock and returns whether it succeded.
- `l.try_lock_for(rel_time)`: Tries to lock the lock with timeout.
- `l.try_lock_until(abs_time)`: Tries to lock the lock with timeout.
- Can change owenership of unique locks: `unique_lock<mutex> l2 = move(l);`.

`std::call_once`

Can be very helpful if we want to do a task only once.
Suppose we have a LogFile class and want to open a file only once:

```
std::call_once(std::once_flag& f, callable [,args])
```

Listening 17.1 : *Synchronization Problems* *cpp*

```
// Problem (1)
if(!f_.open){
  unique_lock<mutex> l(mu_open);
  f_.open();
}
// Problem (2)
unique_lock<mutex> l(mu_open);
if(!f_.open){
  f_.open();
}

//Solution
std::once_flag flag_;

std::call_once(flag_, [&](){f_.open();});
```

(1): If multiple threads enter the if block one will open a file and the other will keep waiting. As soon as the first thread is finished opening the file, the next thread will open a **different** file and so on.
(2): Putting the lock outside the if block is unefficient.

2. Deadlocks

Listening 17.4 : *Thread 1*

```
lock_guard locka(mu1);
// Thread 1 waits for mu2
lock_guard lockb(mu2);
```

Listening 17.5 : *Thread 2*

```
lock_guard locka(mu2);
// Thread 2 wait for mu1
lock_guard lockb(mu2);
```

2.1. `std::lock`

Tries to lock multiple locks at the same time and hence helps to avoid deadlocks

```
std::lock(mu1, mu2 [,...]);
```

Important: if we want to make sure that the locks will get unlocked if we are done we need to use RAII with `lock_guard` and thus have to adopte the lock.

```
  std::lock(mu1, mu2 [,...]);
    std::lock_guard<std::mutex> lock1(mu1,
↪   std::adopt_lock);
    std::lock_guard<std::mutex> lock2(mu2,
↪   std::adopt_lock);
    // access shared data protected by the mu1 and mu2
```

Alternatively we can achieve the same by using `std::defer_lock` and `std::unique_lock`:

```
    std::unique_lock<std::mutex> lock1(mu1,
↪   std::defer_lock);
    std::unique_lock<std::mutex> lock2(mu2,
↪   std::defer_lock);
  std::lock(mu1, mu2 [,...]);
    // access shared data protected by the mu1 and mu2
```

Difference

`std::lock_guard` is cheaper than `std::unique_lock` but if one forgets one of the `lock_guard` statements, the compiler will not show any error and there will be deadlock.
If one forgets one of the `std::unieque_lock` statements the compiler will print an error.

How to avoid Deadlocks

1. Prefer locking single mutexes if possible.
2. Avoid locking a mutex and then calling a user-provided function, as it may try to lock another lock or even the same lock.
3. If you have to lock multiple locks at the same time use `std::lock` function.
4. If you cannot use `std::lock` make sure that you lock the mutexes in the same order in all threads.
5. Alternatively introduce a hierachy of locks.

Listening 17.1 : Examle *cpp*

```cpp
#include <mutex>
#include <thread>
using namespace std;

struct bank_account {
    explicit bank_account(int balance) : balance(balance) {}
    int balance; mutex m;
};

void transfer(bank_account &from, bank_account &to, int
↪ amount) {
    // attempt to lock both mutexes without deadlock
    lock(from.m, to.m);
    /* make sure both already-locked mutexes are unlocked
↪ when
    we're done; => need to adopte the locks to use RAII*/
    std::lock_guard<std::mutex> lock1(from.m, adopt_lock);
    std::lock_guard<std::mutex> lock2(to.m, adopt_lock);

    from.balance -= amount; to.balance += amount;
}
int main() {
  bank_account my_account(100);
  bank_account your_account(50);

  thread t1(transfer, ref(my_account), ref(your_account),
↪ 10);
  thread t2(transfer, ref(your_account), ref(my_account), 5);
  t1.join(); t2.join();
}
```

3. Cache-thrashing

Definition 17.6 **Cache Thrashing:** A thread invalidates the cache of the other threads by writing into a global variable and the variable has to be reloaded by all the other threads.

Listening 17.6 : *C.T.*

```cpp
void sumterms(
long double& sum, size_t i, size_t
↪ j){
  sum = 0.0;
  for(size_t t=i; t<j; t++)
    sum += (1.0-2*(t%2)) / (2*t+1);
}
```

Listening 17.7 : *No C.T.*

```cpp
void sumterms(
long double& res, size_t i, size_t
↪ j){
  long double sum = 0.0;
  for(size_t t=i; t<j; t++)
    sum += (1.0-2*(t%2)) / (2*t+1);
  res = sum;
}
```

Note

- **No C.T.**: Local double `sum` gets allocated on the stack of threads that are on different regions of memory
  ⇒ no longer the same cache line.
- Make sure that theads do not write into the same region of memory.
- Always make scaling plots: Nb. of threads vs. Speedup

4. Condition Variables `std::cond_variable`

Can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition), and notifies the condition_variable.
Thus it helps to avoid busy waiting and allows to notify threads on events just in time.

```cpp
std::condition_variable cond;
cond.notify_(one|all)(); // notifying waiting threads
cond.wait(lock [,predicate]) // wait for notification
```

- The thread that intends to modify the (condition) variable has to:
  1. acquire a `std::mutex` (typically via `std::lock_guard`)
  2. perform the modification while the lock is held
  3. execute `notify_one` or `notify_all` on the `std::condition_variable` (the lock does not need to be held for notification)
- Any thread that intends to wait on `std::condition_variable` has to
  1. acquire a `std::unique_lock<std::mutex>`, on the *same mutex as used to protect the shared condition variable*
  2. execute `wait`, `wait_for`, or `wait_until`.
     The wait operations atomically release the mutex and suspend the execution of the thread.
  3. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

Important Functions of Condition Variables

- `cond.wait_for(lock, rel_time)`: lets lock wait for relative time.
- `cond.wait_until(lock, abs_time)`: lets lock wait until.

Listening 17.8 : *Thread 1*

```cpp
... // E.g. producer of data
cond.notify_one();
...
```

Listening 17.9 : *Thread 2*

```cpp
while(!data){ // e.g. consumer of
↪ data
  unique_lock<mutex> l(m);
  // if(!...){ busy waiting}
  cond.wait(l);
}
```

Why do we need to pass a lock?

The wait function of the condition variable expects a lock because it temporarily unlocks the lock and puts the thread to sleep. This is because a sleeping thread does noot need to lock anything, as it does nothing. Wait will then lock the lock again when it gets notified.

Notes

- Not possible with normal lock guard, as we need to lock multiple times.
- **Spurious Wake**: sometimes it is possible for the lock to wake up by itself, therefore it is good practrice to give the wait function a callable predicate, indicating if the thread was already supposed to wake up.
- `std::condition_variable_any` can be used with any lock and not only with unique locks.

5. Future and Promise

5.1. Future `std::future`

Is a variable that will return an promised value in the future and is not yet filled during initalization.
Thus a future can be used to get the condition variable send over by the promise.

```cpp
using namespace std;
future<returnType> fu [=
↪ (asynch(...)|promise(...)|packaged_task(...));]
...
returnType var = fu.get();
```

Imortant Functions of Futures

- `fu.wait();`: wait untill the child thread is finished.
- `fu.get();` calls wait and then return the value.
- `fu.wait_for(rel_time);`: wait untill rel_time is finished.
- `fu.wait_until(abs_time);`: wait untill abs_time is finished.

5.2. Asynch `std::async`

Is a another way to spawn a thread and returns a future as its value. `std::async` can either spawn a child thread or run in the same thread.

```cpp
std::async((launch_options,) callable (,args))
```

The launch options can be used to force async to cerate a new thread:

1. `std::launch::async`: Forces the creation of a new thread.
2. `std::launch::deferred`: Deferres execution of user given function until get function is called and is then executed in the same thread as the get function.
3. `std::launch::async | std::launch::deferred`: is the default and may or may not creat a new thread.

5.3. Promise `std::promise`

Promise to pass a value from the parent thread to the child thread somtimes in the future

```cpp
std::promise<Type> p;
std::future<Type> fu = p.get_future();
// do somthing else
p.set_value(value);
// Somwhere else
fu.get();
```

Listening 17.10 : *Why do we need futures/promises?* *cpp*

```cpp
void factorial(int N, int& x)
{
  int res(1);
  for(int i=N; i>1; i--)
  res *=i;
  x = res;
}
```

```cpp
int main(){
  int x;
  thread t1(factorial, 4,
↪ std::ref(x));

  t1.join();
  return 0;
}
```

```cpp
using namespace std;
```

The resource x, that shell be returned by the function `factorial` is shared by different threads:

1. Need a mutex to avoid race conditions.
2. May also need a conditon variable to notify when x will be filled.

⇒ the code becomes more inflated and unreadable ⇒ this is where futures come in handy.

Listening 17.11 : *Better solution* *cpp*

```cpp
int factorial(
   std::future<int>&  f)
{
  int N = f.get();
  int res(1);
  ...
  return res;
}

#include <future>
using namespace std;

int main(){
  int x;
  promise<int> p;
  future<int> fu2 =
↪ p.get_future();

  future<int> fu =
↪ async(factorial, 4,
↪ std::ref(fu2));
  // do somthing else
  p.set_value(4);
  x = fu.get();
  return 0;
}
```

The resource x, that shell be returned by the function `factorial` is shared by different threads:

1. Need a mutex to avoid race conditions.
2. May also need a conditon variable to notify when x will be filled.

⇒ the code becomes more inflated and unreadable ⇒ this is where futures come in handy.

Note

- `fu.get();` can only be called once!
- If we forget to fullfil the promise to send over a variable ⇒ `std::future_errc::broken_promise`.
- The child thread tries to do what it can do until my unitl my pormise arrives in the future.
- Neiter future nor promise can be copied, they can only be moved.

5.4. Shared Future `std::shared_future`

Problem: assume we want to call many different threads with the same promised value. This is not possible with a normal promise as `fu.get()` may only be called once ⇒ shared future.

```cpp
using namespace std;
promise<Type> p;
future<Type> fu = p.get_future();
shared_future<Type> sf = fu.share(); // can be passed to
↪ functions
...
Type var = f.get();
```

5.5. Packaged Task `std::packaged_task`

Is a task that can be transported to different palces in the program to be executed. Thus it can be executed in a different place than where it was created.

```cpp
std::packaged_task<returnType(argType)> t(callable);
// somewhere else
t.([args]);
t.get_future().get();
```

Note

- It can be executed in a different place than where it was createdl.
- Can not pass additional parameter into constructor of packaged task ⇒ need to use `std::bind` in order to pass predefined arguments to task:

```cpp
std::packaged_task<returnType()> t
↪ (std::bind(callable,args));
```

Why do we need a packaged task?

Question: could we not simply use a function object?

```cpp
auto t = std::bind(callable,args);
t(); // somwere else
```

Packaged task can link a callable object to a future, which is very important in a threading enviroment.

Listening 17.2 : Example: task handler for packaged tasks *cpp*

```cpp
using namespace std;
deque<packaged_task<Type()> > task_q;
mutex mu;
condition_variable cond;

void task_handler(){
  packaged_task<Type()> t;
  {
    unique_lock<mutex> lock(mu);
    cond.wait(lock, [](){return !task_q.empty();});
    t = move(task_q.front());
    task_q.pop_front(); // remove task from queue
  }
  t();  // run task
}

int main(){
  thread t1(task_handler);
  packaged_task<Type()> t(bind(callable_task, args));
  future<Type> fu = t.get_future();
  {
    lock_guard<mutex> lock(mu);
    // t no longer used in main thread => can use move
    task_q.push_back(std::move(t));
  }
  cond.notify_one(); // new task available
  auto val = fu.get(); // return value of callable task
  t1.join();
}
```

6. Barriers

What if we want for all threads to wait unil some event has happend/they have reached a certain point (e.g. Finite elements calculation)?
Then we need to define a barrier that lets the mutexes wait, based on certain conditions

Add code last page multithreading barrier

Note

Barriers are not built into c++ standard as they are really expensive, but somtimes it is unavoidable: wait for all threads to finish update steps in a Monte Carlo simulation or integration of a PDE.

C++17
C++20

7. **Spans**

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

$$X_{n+1} = (aX_n + c) \mod m \qquad X_0 : \text{(Seed)} \tag{18.1}$$

Corollary 18.1 Quality Of RNG:

1.2. Lagged Fibonacci Generators (LFG)

2. Seeding

3. Non-Uniform Random Numbers

Examples

3.1. Building A Program

Example 18.1
User provided vs declared constructor[def. 8.1]:
Before C++11 aggregates could not have user *declared* constructors but since C++11 it is allowd to have user *provided* constructors in order to declare `default` constructors. See also section about classes.

```
struct Aggregate {
    Aggregate() = default;
};
```

Example 18.2 Equal or Brace Initialization[def. 8.1]:
Since C++11 we can initialize non-static members directly inside the class but this equivalently to writing your own constructor as such it is not possible for a class that wants to be an aggregate:

```
struct NotAggregate {
    Aggregate() = default;
    int v = 5;
    std::vector<double> v{1,2,3};
};
```

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

Example 18.4 Example: Checking for compiler ver-

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

Example 18.5 Function Forward Deceleration[def. 5.9]:

```
float my_fu(float);
my_fu(8);
float my_fu(float var){ return  var*2};
```

Example 18.6 Class Forward Deceleration[def. 5.9]:

```
class MyClass;
MyCLass *ptr;
do_something(ptr)
definition somewhere else
```

Example 18.7 Global Scope Variable[def. 5.14]:

```
Type var;
int main(){
  do_something()
}
```

stack hep initalization

Example 18.8 Template Aliasing[cor. 8.2]:

```
template<class T>
using ptr_t = T*;
ptr_t<int> my_ptr;
```

Functions

Example 18.9 Static function variables[def. 8.44]:

```
void counter(){
  static int count = 0;                    only used once
  std::count << ++count;
  // 1, 2, 3,...
}
```

Example 18.10 Return By Reference: Allows us to modify the return value of a function:

```
int& getLargest(int arr[], int size){
int indexOfLargest = 0;
for(int i=1; i < size; i++){
    if(arr[i] > arr[indexOfLargest]){
    indexOfLargest = i;
    }
}
return arr[indexOfLargest];
}
```

Compex Data Types

Example 18.11 Enum:

```
class Car{
public:
    enum Color{ Red, BLUE, WHITE };
    void setColor(Car::Color color){
        _color = color;
    }
    Car:Color getColor() const
    {
        return _color;
    }
private:
    Car::Color _color;
}
```

Example 18.12 Reading everything from cin:

```
    cout << "Enter a number (-1 = quit): ";
    if (!(cin >> input_var)) {
        cout << "Please enter numbers only." << endl;
        // Clear stream state
        cin.clear();
        // Ignore previous input
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }
} while (input_var != -1);
```

Initialization

Example 18.13 Default Initialization[def. 8.56]:

```
struct MyStruct
{
    default initi/garbage
    int i;
    zero init/empty string=""
    std::string s;
    default initi/garbage
    int *ptr;
}
```

```
int main{
  MyStruct ob;
}
```

Example 18.14 Value Initialization[def. 8.57]:

```
struct MyStruct
{
    zero initi=0
    int i;
    zero init/empty string=""
    std::string s;
    zero initi=nullptr
    int *ptr;
}
```

```
int main{
  MyStruct ob{};
}
```

Classes

Example 18.15 Mutable[def. 9.15]:

```
MyClass{
    ...
    void myFun(){
        ...
        _cnt++;
    }
    ...
    mutable size_t _cnt;
};
```

Example 18.16 Uneccessary Assignment:
1. T ctor called for a
2. copy assig. ctor called for v_
3. dest called for a

```
class MyClass {
    T v_;
public:
    MyClass(T a) {
        v_ = a;
    }
};
```

1. T's copy ctor called for a
2. dest called for a

```
class MyClass {
    Type v_;
    public:
      MyClass(T a):v_(a) {}
};
```

Example 18.17 Direct Variable Initialization:

```
class MyClass{
  public:
  MyClass(T a1, T& a2): cv_(a1), rv_(a2){
    assignment here not possible
  }
    const T cv_;
    T& rv_;
}
```

Example 18.18 In Class Initialization[def. 9.11]:
Member Init. List

```
struct Widget {
    Widget()
    : i {}
    , s {}
    , pi{}
    {}
    Widget( int j )
    : i {j}
    , s {}
    , pi{}
    {}
    int i;
    std::string s;
    int* pi;
};
```

In class member init.

```
struct Widget {
  Widget() = default;
  Widget(int j)
  : i{j}
  {}
  // In-class initializers
  val 8 used for default
  int i{8};
  std::string s{};
  int* pi{};
};
```

3.1.1. Operators

Example 18.19 Right Associacity[def. 9.21]:
$a = b = c = d \qquad \widehat{=} \qquad a = (b = (c = d))$

Example 18.20 Symmetric Operators[def. 9.27]:

```
Point operator+(const Point &p1, const Point &p2){
  Point newPoint(p1.x()+p2.x(), p1.y()+p2.y());
  return newPoint;
}
```

or even more efficient as:

```
Point operator+(const Point &p1, const Point &p2){
  return p1+=p2; usingexample 18.21
}
```

Example 18.21 Assignment Operators[def. 9.25]:

```
Class Point{
public:
    const Point &operator+=(const Point& rhsPoint){
        x_ += rhsPoint.x_;
        y_ += rhsPoint.y_;
        return *this;
    }
private:
    double x_{};
    double y_{};
}
```

Example 18.22 Functors: class MyFunc public: MyFunc(double l): lambda(l) double operator() double(x) return exp(-lambda*x); private: const double lambda;

3.1.2. Inheritance

Example 18.23 Dynamic vs. Static Binding:

```
class Animal{                                          headr.hpp
  void eat(){ std::cout << "I'm eating generic food."}
};
class Cat : public Animal{
  void eat(){ std::cout << "I'm eating a rat."}
};
void whatrDoYouEat(Animal &genericAnimal){
  genericAnimal->ear();
}
int main(){                                            main.cpp
  Animal animal{};
  Cat cat{};
  animal->eat(); // I'm eating generic food.
  Cat->eat(); // I'm eating generic a rat.
  whatrDoYouEat(animal); // I'm eating generic food.
  whatrDoYouEat(cat); // I'm eating generic food.
}
```

Example 18.24 Abstract Class[def. 9.48]:

```
class Shape{
  public:
    virtual int getArea() = 0;
    void setWidth(int w){ width = w; }
    void setHeight(int h){ height = w; }
  protected:
    int width{};
    int height{};
}
class Triangle : public Shape{
  public:
    override int getArea(){
      return 1.0/2*height*width;
    }
}
```

Example 18.25 Aging of derived Classes[def. 9.49]:
Aging of different types of animals:

```
void age(std::vector<Animal*> a){
  for(int i=0; i<a.size(); i++)
    a[i]->grow();
}
```

Tempaltes

Example 18.26 Dependent Names:

```
template <class T>
dependent_name<T>;
T::dependent_name;
```

Exception Handling

Example 18.27 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 18.28 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";
  }
}
```

Algorithms and Data Structure

Example 18.29
Creating a sequence of exponents[def. 16.1]**:**
Create sequences of $2^1, 2^2, \ldots, 2^5$:

```
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);}
);
```

Title — Listening 18.1

```
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;
}
```

Example 18.30 for Loop Iteration:

```
container<type>::iterator it = [pos];
container<type>::iterator endPos = [endpos];
for(; it != endPos; it++)
   do something
```

Example 18.31 While Loop Iteration:

```
container<type>::iterator it = [pos];
container<type>::iterator endPos = [endpos];
while(it != endPos){
   do something
   it++;
}
```

Example 18.32 Priority Quque Min/Max Heap[def. 14.18]**:**

```
#include <queue>
std::priority_queue<int> max_q;
std::priority_queue<int, std::vector<int>,
↪  std::greater<int>> min_q;
```

3.1.3. Lists

Example 18.33 Push Back[def. 14.10]**:**

```
void push_back(data_t data){
    if(!head)
        head =  new Node(data);
    else{
        Node *curr = head;
        while(curr->next)
            curr = curr->next;
        curr->next = new Node(data);
    }
};
```

or alternatively we check if the pointer is a `nullptr` and if not we change the address of the current pointer by using a temporary pointer pointing to the pointer:

```
void push_back(data_t data){
    Node** current = &head;
    while(*current != nullptr)
      current = &(*current)->next;
    *current = new Node(data);
}
```

Example 18.34 Push Front[def. 14.10]**:**

```
void push_back(data_t data){
    if(!head)
        head = new Node(data);
    else{
        Node *curr = head;
        head = new Node(data);
        head->next = curr;
    }
};
```

Example 18.35 Destructor[def. 14.10]**:**

```
~Node(){
    Node* curr = head;
    Node* next;
    while(curr) {
        next = curr->next;
        delete curr;
        curr = next;
    }
    head = nullptr;
 };
```

C++11

Example 18.36 Unique Pointer[def. 8.30]**:**

```
void do_something() {
 std::unique_ptr<int[]> data = std::make_unique<int[]>(10);
}                                          automatic deallocation
int main(){ do_something();}
```

Example 18.37 Unique Pointer and Copying[def. 8.30]**:**
Unique pointers cannot be copied or passed:

```
void do_something(std::unique_ptr<int[]> p) {...}
int main() {
 std::unique_ptr<int[]> ptr = std::make_unique<int[]>(42);
 std::unique_ptr<int[]> ptr_copy = ptr;                  Error
 do_something(ptr);       Error ptr cannot be passed by copy
}
```

Example 18.38 Multithreaded Hello World:

```
#include <iostream>
#include <thread>

void hello(){
 std::cout << "hello\n";
}

int main(){
 std::thread t(hello);
 // wait for termination of t
 t.join();
 return 0;
}
```

Example 18.39 Multithreaded Hello World With Parameters:

```
void hello(int id){
  std::cout << "hello from " << id << "\n";
}
int main(){
  std::vector<std::thread> tv(3);
  int id = 0;
  for (auto & t:tv)
    t = std::thread(hello, ++id);
    std::cout << "hello from main \n";
    for (auto & t:tv)
      t.join();
    return 0;
}
```

Example 18.40 Callable Objects:
Let: `foo` be a function and `c` an object of a functor class `C` with the member function `m1` and `m2`.
Question: what kind of objects can we pass as callable objects?

- Passing a normal functino:
  `std::thread t1(foo [,args]);`
- Copy of `c` in a different thread:
  `std::thread t1(c [,args]);`
- `c` executed in a different thread:
  `std::thread t1(std::ref(c) [,args]);`
- If we don't want to make a copy of `c` but also dont want to pass it as reference either, we can move `c` from the Parent thread to the child thread:
  `std::thread t1(std::move(c) [,args]);`
  Attention: `c` will no longer be usable in the parent thread.
- Passing a temporary/rvalue:
  `std::thread t1(C() [,args]);`
- Passing lambda functions:
  `std::thread t1(lambda_function [,args]);`
- Passing a member function of a copy of `c`:
  `std::thread t1(&C::m1, c, [,args]);`
- Passing a member function of a refernce of `c`:
  `std::thread t1(&C::m1, &c, [,args]);`

4. Implementations

Implementation 18.1 Iterator Idea[def. 13.1]:

```
template<class T>
class Container {
  public:
    typedef T* iterator;
    Container();
    Container(size_t);
    Iterator begin() {return p_;}
    Iterator end() {return p_+size_;}
  private
    T* p_;
    size_t size_;
}
```

C

Functions

Vardic Functions #include⟨stdarg.h⟩

Definition 18.2 Vardic Functions:
Vardic functions are functions that can take an variable number of arguments indicates by thee dots:

```
myfu([arg1, arg2], ...)
```

5.1.1. Types

Type 18.1
Vardic List:
Is a data types that is able to hold a variable number of arguments:

```
va_list a_list;
```

5.1.2. Functions/Macros

Va Start

Function 18.1 Va Start:
Initializes the a `va_list` object and expects a `va_list` and the last fixed argument of our function, usually the number of arguments:

```
myfu(int num_args, ...){
    va_list l;
    va_start(l, num_args);
}
```

Va Arg

Function 18.2 [example 18.41]
Va Arg:
Retrieves the next variable argument, given a `va_list` and its data type:

```
type var = va_agr(l, type)
```

Va End

Function 18.3 [example 18.41]
Va End:
`va_end` must be called before we can use any of the retrieved arguments. It also resets the pointer of the `va_list` object back to `NULL` a `va_list` and the last fixed argument of our function, usually the number of arguments:

```
va_end(l);
```

Va Copy
Vasprintf

C Examples

Example 18.41 Vardic Arguments:

```
myfu(int num_args, ...){
    va_list l;
    va_start(l, num_args);
    for(int i = 0; i < num_args; i++)
        do_something(va_arg(l, type))
    va_end(l);
}
```