RUHR-UNIVERSITÄT BOCHUM

RUB

Scientific C++ Programming

Prof. Dr. Andreas Vogel, Poria Saberi

Content

Introduction

Overview

C++

C++ is

- an extension of the language C
- syntactically very similar to C / Java / C#
- an object oriented programming language
- also a functional programming language
- a typed language

History

A brief history of C++:

- 1980: Development by Bjarne Stroustrup
- 1883: release of the first public version
- 1998: C++98 - ISO/IEC/ANSI/DIN - standardization
- 2003: C++03 - minor update and fixes
- 2011: C++11 - major revision
- 2014: C++14 - minor improvements
- 2017: C++17 - major revision
- 2020: C++20 - major revision
- 2023-Present: C++23 - the next version

Compiler and linker

Compiler

- A C++ source code is **just** plain text, meaning it is
  - human readable
  - editable by any text editor
- Computers can **not** understand the source code directly; therefore,
  - a **translator** called **compiler** is required
- The compiler does two things:
  - Makes sure that the source code follows the C++ rules
  - Translates the source code into machine-readable units, called **object** files

  Several free and paid compilers are available, e.g.,
  - g++ from the GNU compiler suite
  - icc (Intel),
  - XL C++ (IBM)
  - clang++ (LLVM project)
  - ...

Linker

A separate program called a **linker** follows the compilation step in order to produce an executable program
The linker

- Combines all the object files
- Links external libraries
- Resolves cross-file dependencies

The output of the linker is an executable program
The program can be executed on any machine running the same architecture

Library

A **library** is a packaged piece of compiled code

Static library:

- The linked library is copied into the binary during the linking process
- The library has to be compiled
- The binary works as a standalone file
- If the linked library is changed, the binary has to be compiled and linked again
- Although distribution is facilitated, the binary size can become too large

Dynamic (shared) library:

- The library is linked during runtime
- The library file(s) has to be present during execution
- If the linked library is changed, no recompilation is necessary as long as the interface remains compatible
- The object files must be position independent, e.g., using the `-fPIC` flag in g++

The C++ Standard Library

An extensive library distributed with C++ that provides a set of common functionalities. Almost all C++ programs use the standard library
The standard library includes:

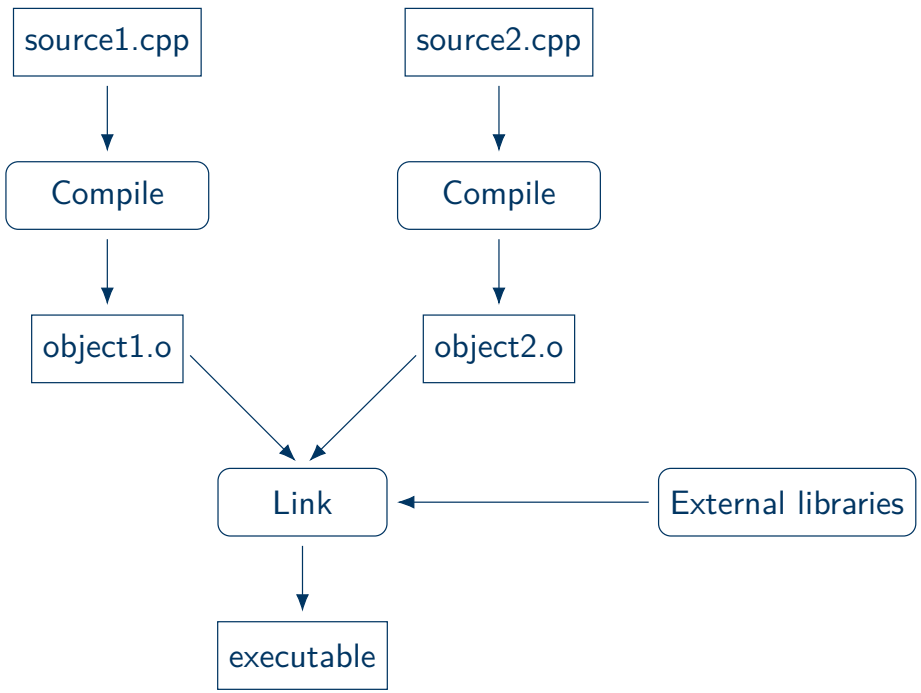
- input/output (I/O) for the console and files
- algorithms, such as sort, search, shuffle, etc.
- data structures, e.g., vector, set, map, list, queue, stack, array, etc.
- filesystem library
- memory management
- multi-threading
- time facilities
- ...

The standard library places requirements not only on the interface, but also the performance of generic algorithms, mostly $O(n)$ or $O(n\,logn)$ time

Development workflow

The production of a C++ program, often referred to as **building**, follows the steps below in its simplest form:

- Implementation of the program source code
- Compilation of the source code into object file(s)
- Liking the object file(s) and potentially external libraries into an executable program



File extensions

The following conventions are typically used for C++ applications:

File type	Usage
.hpp / .hh /.h	C++ header file
.cpp / .cc	C++ source file
.o	object file
.a	static library
.so	dynamic library
.h	C header file
.c	C source file

Compiler options

The behavior of every compiler can be configured, often intricately, using **options**

Know Thy Compiler!

Options for the g++ compiler start with "–".

Some important options:

- `-c`: compile, i.e. create object file(s)
- `-o`: link, i.e. create an executable
- `-std=c++[xx]`: use a specific standard
- `-O0` ... `-O3`: specify the optimization level
- `-g`: debug mode
- `-Wall`: show all warnings

Hello World!

A first program

Source code:

```cpp
#include <iostream>

int main() {
  std::cout << "Hello world!\n";
  return 0;
}
```

hello_world.cpp

Output:
Prints "Hello world!" to the screen
Steps:

- Compilation
- Linking
- Execution

Compilation

Creating object file(s):

```
$ g++ -c hello_world.cpp
$ ls
  hello_world.cpp hello_world.o
```

source code (*.cpp, *.cc) ⇒ **object file(s)** (*.o)

One source file is translated into **one** object file containing executable commands.

There may be many object files.

```
#include <iostream>

int main() {
 std::cout<<"Hello world!";
 return 0;
}
```

hello_world.cpp

⇒

```
a_atexit_ZNSt8ios_base4Init
sonality_v0__ZSt3minImKT
e_main__GLOBAL__main
verify_groupingPmRKSsC0
_dsfgdf_susne
```

hello_world.o (hex-file)

Linking

Create an executable program:

```
$ g++ -o hello_world hello_world.o
$ ls
  hello_world.cpp hello_world.o hello_world
```

object files (*.o) and **libraries** (*.so, *.a) $\Rightarrow$ **executable**

Several object files and libraries are combined into **one** executable program

Assembler code

Creating assembler code:

```
$ g++ -S hello_world.cpp
$ ls
  hello_world.cpp hello_world.s
```

source code (*.cpp, *.cc) $\Rightarrow$ **assembler code** (*.s)

A source file is translated into assembler code (machine instructions).

```
#include <iostream>

int main() {
 std::cout<<"Hello world!";
 return 0;
}
```

hello_world.cpp

$\Rightarrow$

```
LCFI5:
    movq    -8(%rbp), %rax
    movq    (%rax), %rax
    cmpq    %rax, %rdx
    jmp     L4
```

hello_world.s

Structure and syntax

Basic structure

- A C++ program consists of a sequence of **statements**
- Statements are often grouped in **functions**
- Statements are executed in order, top to bottom
- There are several types of statements in C++:
  - Declaration statements
  - Expression statements
  - Compound statements
  - Iteration statements
  - Selection statements
  - Labeled statements
  - Jump statements
  - Try-catch blocks

Where everything starts

Every C++ program starts at a function with the name `main`:

```
int main() {
 // starting here...
}
```

main.cpp

In every program, there must be **exactly one** main function.

Formatting

- Formatting refers to how the source code looks like
- Whitespace, i.e., formatting characters such as spaces are mostly ignored
- C++ does **NOT** enforce a rigid formatting, meaning there is no single right way to format the source code
- Nevertheless, it is important to write **clean** and **legible** code
- There are several formatting conventions. Choose **one** and be **consistent**
- See the C++ Core Guidelines

Formatting

Choose **one** and be **consistent**

- Both tabs and spaces are fine for indentation; however, spaces are typically preferred because of universal compatibility
- Each line should be split at maximum 80 characters long
- Curly brackets should be formatted consistently

```
int main() {
    // body
}
```

```
int main()
{
    // body
}
```

- Code blocks within curly brackets should be indented one level
- Whitespace should be used to make the code easier to read

```
int long_name = 4;
int x = 5;
double y = 4.5;
```

```
int long_name = 4;
int x         = 5;
double y      = 4.5;
```

Comments

Comments

- are pieces of information for the human reader of the code
- are ignored by the compiler
- are used for the documentation of the code

Best practice

- Use comments as much as possible!
- Use a consistent style!
- Comments should add value to the code, i.e., do NOT duplicate what is already clear
- Good comments are NOT a replacement for bad code
- Explain esoteric code, debugs and generally what may not be immediately clear

Comments

Comments can be realized in two ways in C++:

- Single-line comments using //
- Multi-line comments using /* ... */

```
int main() {
 int x = 5; // ... this is a single-line comment
 int y = 6; /* another comment */ int z = 7;
 /* comments can
  spread
  several lines */
}
```

Declaration and definition

- A **declaration** tells the compiler about the existence of an identifier and its associated type information without the need to necessarily provide the definition
- A **definition** is a declaration that implements or instantiates the identifier
- Therefore, while all definitions are declarations, not all declarations are definitions
- **The one definition rule (ODR)** loosely states that each identifier must have one definition within a program
- Violation of the ODR results in a compiler/linker error or undefined behavior

Header files

Large C++ programs are virtually always implemented in multiple files.
There are two major file types:

- Source files, typically with a .cpp/.cc extension
- Header files, typically with a .h/.hpp/.hh/[-] file extension

Source files typically include **definitions** while header files include **declarations**.
Such division allows for reusing header files in multiple locations and avoiding the need for re-declarations.

Best practice

The .h file extension should typically be preferred.
The source and header file pairs are often named the same, e.g., `some_code.cpp` and `some_code.h`.

All will become clear when we discuss modular programming design.

Preprocessor

- The **preprocessor** is a program that modifies the source code prior to compilation in various ways
- The actual source files are **NOT** modified
- Preprocessing directives include
  - `#include <some_file>`: replaces the the directive with the content of the file
  - `#define`: defines object-like and function-like **macros**
  - `#if`, `#ifdef`, `#ifndef`, `#endif`: defines conditional compilation
- The output of preprocessing can be generated using the `-E` option in g++

#include

The #include directive copies the content of the specified file to the code position during preprocessing.

```
#include "some_code.h"
int main() {
 std::cout<<"Hello world!";
}
```

hello_world.cpp

```
struct SomeCode {};
```

some_code.h

```
  $ g++ -E hello_world.cpp
 [ ... ]
struct SomeCode {};
# 2 ‘‘hello_world.cpp‘‘ 2
int main() {
 std :: cout << ‘‘Hello world!\n‘‘;
}
```

#include

The #include directive is typically used to import header files.
The imported declarations can then be used within the file that imports them.
Reminder: the **definitions** will be resolved during **linking**.

Example: importing the standard I/O library for data input/output

```
#include <iostream>  // include the standard I/O library

int main() {
  // std::cout from the I/O library can be used
    std::cout << "Writing ...";
    std::cout << "... next line " << std::endl;
    std::cout << "Can " << "be " << "concatenated ...";
    std::cout << "Or numbers such as: " << 4 << " are written";
    int value;
    std::cin >> value;  // std::cin from the I/O library can be used
}
```

Variables and data types

Variable

- A **variable** is a named place in memory used for storing **data**
- The **identifier** (name) allows us to refer to variables
- A valid identifier is a sequence of one or more letters, digits and the underscore character that obey certain rules, e.g.,
  - Other characters such as spaces, symbols and punctuation marks cannot be a part of identifiers
  - An identifier cannot start with a digit
  - reserved C++ keywords, such as `for`, `true`, `if`, `etc.` cannot be used as identifiers
- C++ is **case sensitive**, i.e., `a` and `A` are different identifiers
- C++ is a **typed language**, i.e., every variable **must** have a type at **compile time**

Identifier conventions

A good naming convention improves the clarity and legibility of the code.

Best practice

- **Be Consistent!**
- Use descriptive identifiers, e.g., `speed_of_light` instead of `c`
- Separate multi-word variables consistently, e.g., `SolverRuntime` or `solver_runtime`
- Avoid naming conflicts, e.g., the same name in nested scopes
- Avoid contractions when possible, e.g., `hidden_variable` instead of `hid_var`
- **Be Consistent!**

Variable declaration

- Variables are **declared** as follows:
  `<data_type> <identifier>;`
- Note that the **declaration statement** must end with a semicolon
- Multiple variables can be declared in the same statement using commas:
  `<data_type> <identifier_1>, <identifier_2>, ...;`
- Once declared, the compiler reserves sufficient memory for variables within their **scope**
- The scope of a variable is the portion of the program where the variable is valid, more on scopes and life cycles later

```
int a; // Declaring a variable of type int with identifier a
int b, b1, b_1, _ba;  // declaring multiple variables
int 1b;               // Invalid identifier
int b#;               // Invalid identifier
int a b;              // Invalid identifier
```

Variable initialization

- Declared variables have an **undefined value**. Beware!
- **Initialization** is when a value is assigned to a variable for the first time
- There are three ways to initialize variables in C++:
  - **C-style initialization** (C/C++):
    `<data_type identifier = value;`
  - **Constructor initialization** (C++):
    `data_type identifier(value);`
  - **Uniform initialization** (C++11):
    `data_type identifier{value};`
  - An empty list in uniform initialization can be used to initialize the variable to its default value

```
int a;      // uninitialized
int b = 2;  // C-style initialization
int c(3);   // Constructor initialization
int d{4};   // Uniform initialization
int e{};    // default value (0 for int)
```

auto and typeid

C++11

C++11 introduced the auto keyword which allows the type of a variable to be automatically deduced during its **declaration** from its **initialization** value.
Variables declared with auto **must** be initialized during declaration.

The typeid operator can be used to get information about the data type of variables and expressions at **runtime**.

```
int a  = 4;
auto b = a;  // b is an int
auto c;      // Invalid. The data type not clear during declaration
c      = 1;
std::cout << typeid(b).name()  // Prints 'i' for integer
```

Native types

Type	Bytes*	Content	Values
`void`	-	no type	
`char`	1	character	'A', 'b', ...
`bool`	1	boolean	`false, true`
`short (int)`	2		$-2^{15} \dots 2^{15}$ $(= 32768)$
`int`	4	integer	$-2^{31} \dots 2^{31}$ $(\approx 2 \cdot 10^9)$
`long (int)`	8		$-2^{63} \dots 2^{63}$
`float`	4		$\pm 1.2 \cdot 10^{-38} \dots \pm 3.4 \cdot 10^{38}$
`double`	8	floating point	$\pm 2.2 \cdot 10^{-308} \dots \pm 1.8 \cdot 10^{308}$
`long double`	12		$\pm 3.4 \cdot 10^{-4932} \dots \pm 1.2 \cdot 10^{4932}$

*) compiler dependent

**) integer also as `unsigned` (e.g. `unsigned int`).

Size of data types

The `sizeof` operator returns the size of a data type in bytes.

```
#include <iostream>
int main(){
    std::cout << "Size (int): " << sizeof(int) << std::endl;
    std::cout << "Size (double): " << sizeof(double) << std::endl;
}
```

Data type limits

The library `limits` provides the limiting values for data types.

```
#include <iostream>
#include <limits>
int main(){
    std::cout << "Max (int): " << std::numeric_limits<int>::max() <<
    std::endl;
    std::cout << "Min (int): " << std::numeric_limits<int>::min() <<
    std::endl;
}
```

Constant variables

The const keyword, a variable can be defined as **constant**.
The value of constant variables cannot be changed after initialization.

```
int main(){
 float Pi = 3.1415926;
 Pi = 3;  // Valid, Pi can be changed

 const float pi = 3.1415926;
 pi = 3;  // Invalid, pi can not be changed
}
```

Best practice

Use const as much as possible!
Using const can prevent errors at **compile time**.

Operators

Assignment

The (plain) = symbol assigns a value.

```
...
int a, b, c;
a = b = c = 5;          // assigns the value
a = (b = (c = 5));      // the same
c = 5; b = c; a = b;    // the same
...
```

There is a subtle difference:

An **initialization** assigns a value at the variable creation time.
An **assignment** assigns a value to an existing variable.

Arithmetic operators

Op.	description	example
+	addition	i + 5
–	subtraction	7 – j
*	multiplication	8 * 4
/	division	9 / 3
%	modulo	5 % 2

Compound assignment operators:

Op.	example	long form equivalent
+=	i += 3	i = i + 3
–=	i –= 3	i = i – 3
*=	i *= 3	i = i * 3
/=	i /= 3	i = i / 3

Comparison operators

Op.	description	example
>	greater	a > b
>=	greater equal	a >= b
<	smaller	a < b
<=	smaller equal	a <= b
==	equal	a == b
!=	not equal	a != b

Floating point comparison

While relative comparison of floating point numbers, e.g., `<`, `>`, `etc.` is safe, because of their internal representation, equivalence comparison, i.e., == and != between floating point numbers is in general not safe.

Logical operators

Op.	description	example
!	logical negation	`!(3 < 4) // false`
&&	logical AND	`(3 > 4) && (3 < 4) // false`
||	logical OR	`(3 > 4) || (3 < 4) // true`

Info

The right-hand side operand of the AND/OR operators is only evaluated if the result is not already determined by the left-hand operand.

Increment/Decrement operators

The ++ and -- operators increase and decrease their operands, respectively.
In the prefix variant, the value of the variable is first changed and then evaluated.
In the postfix variant, the value of the variable is evaluated first and then changed.

```
int a = 5, b = 5, c = 5;
c++;           // now: c = 6
int d = ++a;   // now: d = 6, a = 6
int e = b++;   // now: e = 5, b = 6
```

Performance tip

The postfix variant creates an internal copy while the prefix variant avoids this overhead. Thus, the prefix variant is faster and should be preferred whenever possible.

Mathematical functions

Many mathematical functions for `float/double` can be found in the library `cmath`

`sqrt(x), exp(x), log(x), pow(x,y), fabs(x), fmod(x), ceil(x), floor(x), sin(x), cos(x), tan(x), asin(x), acos(x), atan(x)`

and some constants: `M_E` $= e$ and `M_PI` $= \pi$.

```
#include <iostream>
#include <cmath>
int main(){
 std::cout << std::sin(2*M_PI);
}
```

Implicit type conversion

Implicit or automatic type conversion is done automatically by the compiler when a variable of one type is assigned a value from a different type.
Such conversion is possible, only if it is meaningful and allowed in C++.
For numbers, such a conversion is always possible, if no information loss occurs, specifically going from a smaller data type to a larger one, e.g., `int` to `double`.
Implicit conversion is prone to information loss, e.g., when going from a larger data type to a smaller one, e.g., `double` to `int`, and **may** lead to a warning.

```
int main(){
  int a = 1;
  double b = 4.99;
  double c = a; // implicit conversion. ok, no data loss
  int d = b;    // implicit conversion to 4. Data loss, no warning
  int e {b};    // implicit conversion to 4. Data loss with warning
}
```

Lifetime and scope

Blocks and lifetime

Blocks

- A **block** is a compound of statements
- A block is created using curly braces { }
- A block can always be used where a statement is admissible
- A block does not require a closing semicolon
- Blocks can be nested

```
// some statements ...
{ // beginning of block
  // some statements ...
  {  // nested block
    // more statements ...
  } // end of block
} // end of block
```

Best practice

Increase indentation by one level in blocks.

Lifetime of variables in blocks

- The compiler allocates sufficient memory for variables at the time of **declaration**
- The **Lifetime** of a variable is the region of the code where it is valid
- **Compile-time** memory allocation is referred to as **static memory** allocation, more on memory allocation in memory management
- The lifetime of variables declared within a block is limited to the block

```
{  // beginning of block
  int a = 4;  // memory for an integer allocated by compiler
}  // memory released here, a is invalid
```

Visibility of variables in blocks

- A variable is only valid after declaration and within the same block
- Within a block each variable can only be declared once
- Variables remain visible in inner blocks
- Inner block variables can hide outer block variables

```
{ // beginning of block
  int a = 4; int b = 4;
  { // inner Block
    int a = 3;
    std::cout << a << b; // a = 3, b = 4;
  }
  std::cout << a << b; // a = 4, b = 4;
} // end of block
```

Namespaces

Namespace

Namespaces are used to group names.

- Used to avert naming conflicts.
- Identical identifiers can occur in large projects, especially when external libraries are used.
- Grouping via `namespace <nspace> {...}`.
- All identifiers read `<nspace>::<name>`
- Have to be located at the global level or inside another namespace.
- Important namespace: `std` (standard library)

Namespace

```
namespace MyNameSpace1 {
 int a = 2;
}

namespace MyNameSpace2 {
 int a = 3
}

int main(){
 int a = 4;
 std::cout << "a: " << a; // a = 4
 std::cout << "a: " << MyNameSpace2::a; // a = 3
 std::cout << "a: " << MyNameSpace1::a; // a = 2
}
```

`using`-Declaration and `using`-Directive

A lot of coding effort can be saved using the `using` directive.

This is possible for a namespace as a whole:

```
using namespace std;
```

or for selected parts only:

```
using std::cout;
using std::endl;
```

`using`-Declaration and `using`-Directive

The `using` declaration makes an identifier available in the respective block without using the scope operator:

```
{
 using std::cout;
 cout << "Hello World" << std::endl;
}
```

The `using` directive makes all names from a namespace available:

```
{
 using namespace std;
 cout << "Hello World" << endl;
}
```

Control Statements

Control statements

Control statements allow us to control the flow of the program depending on logical conditions and come in two forms:

- Conditional statements
  - `if` and `if-else` statements
  - `switch` statements
- Loops
  - `for` loop
  - `while` loop
  - `do-while` loop
  - `break` and `continue`

Control statements are based on logical expressions that evaluate to a boolean value.
Logical expressions are constructed with the help of boolean operators, e.g., >, <, ==, !=, &&, ||, etc.

Control statements

Control statements have the following general syntax:

```
<control statement> ( <statements/expressions> )
{
    <statements>
}
```

where `<control statement> ( <expressions> )` defines the control statement and `<statements>` defines the body of the control statement. Note that the brackets `{ }` are often optional when there is a single statement to be executed.

Best practice

Choose a style for each control statement and be consistent!

Conditional statements

`if` statement

The conditional `if` statement has the following syntax:

```
if ( <logical expression> ) {
    <statements>
}
else if ( <logical expression> ) {
    <statements>
}
else {
    <statements>
}
```

`if` statement

`if-else if` blocks have the following properties:

- `else if` and the `else` block are optional
- At most, one `else` block can be defined
- The statements for the corresponding `if`/`else if` is executed if the corresponding logical expression is evaluated to `true`
- Each block is mutually exclusive and at most one block will be executed
- As soon as one block is executed, the subsequent blocks are skipped, i.e., the logical expressions of the remaining blocks will not be evaluated

if statement

```
if (a > 5) a = 5;
else if (a > 4) a = 4;
else a = 3;

if (a > 5) {
  a = 5;
} else if (a > 4) {
  a = 4;
}
else {
  a = 3;
}
```

`switch` statement

The `switch` statement has the following syntax:

```
switch ( <statement> ) {
    case <constant 1>:
        <statements>
        [break;]
    ...
    case <constant n>:
        <statements>
        [break;]
    default:
        <default statements>
}
```

`switch` statement

The `switch` statement has the following properties:

- `<constant statement i>` should be a **compile-time** expression that can evaluate to an integer or character
- Everything after the first matching `case` label is executed unless a `break` statement or the end of the `switch` block is reached
- Although `break` statements are optional, they must be used when needed to produced the desired flow
- Once a `break` statement is reached, the `switch` statement is left immediately
- If none of the `case` labels are matched, the `default` label is executed

switch statement

```
int main(){
 using namespace std;

 int a;
 cin >> a; // user inputs an integer

 switch(a)
 {
    case 1 : cout << "User typed a 1" << endl; break;
    case 2 : cout << "User typed a 2" << endl; break;
    case 3 : cout << "User typed a 3" << endl; break;
    default : cout << "User typed a character != {1,2,3}" << endl;
  }
}
```

if-else and switch

Repeated `if` and `else if` statements can reproduce a similar and more flexible structure to `switch` statements.

```
if (a >= 0 && a < 10)
 cout << "0 <= a < 10" << endl;
else if (a >= 10 && a < 20)
 cout << "10 <= a < 20" << endl;
else if (a >= 20 && a <= 30)
 cout << "20 <= a <= 30" << endl;
else
 cout << "a is not in the range [0,30]" << endl;
```

Loop statements

`for` statement

The `for` loop has the following syntax:

```
for ( <init>; <condition>; <increase> )
     <statement>
```

- The init statement is used to declare and/or initialize variables and is executed first and only once
- The scope of the variables declared in the init statement is limited to the `for` block
- The condition is evaluated next, and the body of the loop is executed if the condition is true
- After the execution of the body, the increase statement is executed and the condition is evaluated again
- The init, condition and increase statements are optional

```
for(int i = 0; i < 10; ++i){
  std::cout << i << std::endl;
}
```

`while` loop

The `while` loop has the following syntax:

```
while ( <logical expression> )
      <statements>
```

- The logical expression is evaluated first, and the body of the `while` loop is executed if it is true
- Note that in contrast to the `for` loop, there is no init statement in the `while` loop; therefore, the loop control variables must be declared beforehand
- Note that in contrast to the `for` loop, there is no update statement in the `while` loop; therefore, the update of the control variables must be done in the body of the loop

```
int i = 0;
while( i < 10 ) {
  std::cout << i++ << std::endl;
}
```

`do-while` loop

The do-while-loop has the following syntax:

```
do
     <statement>
while ( <logical expression> );
```

- The `do-while` loop has a similar structure to the `while` loop with the difference that the logical expression is evaluated first after the execution of the loop body, i.e., the body is executed at least once
- Similar to the `while` loop, there is no init or update statements in the `do-while` loop

```
int i = 0;
do {
  std::cout << i++ << std::endl;
} while ( i < 10 );
```

break and continue

`break` and `continue` can be used to manipulate the program flow

- `break`: The program control immediately leaves the `switch`, `while`, `do-while`, `for` statement
- `continue`: stopping current cycle and continuing the next cycle in `while`, `do-while`, `for` statements.

```
int i = 0;
while( true ){
 if( i == 10 ) break;
 std::cout << i++ << std::endl;
}

for(int i = -10; i < 10; ++i){
 if( i < 0 ) continue;
 std::cout << i << std::endl;
}
```

References, Pointers and Arrays

References

Reference

A **reference** is an alias for a variable. The reference can be used instead of the original variable. This way, a variable can have different names.

C++ references are realized via a trailing &-sign to the data type.

```
...
int a = 1;
int& b = a; // reference of a
b = 3; // a is now 3
std::cout << "Value for a: "<< a << std::endl;
...
```

const-Reference

References can be declared constant. The value of a const reference can not be changed. Thereby, one can enforce to grant only read access.

```
...
int a = 1;
const int& b = a; // const-reference of a
a = 2; // Ok: a is not const
b = 3; // Error: reference is const
...
```

Pointers

Pointer

Accessing variables is not only possible directly, but **pointer** can be used. A pointer points to the memory location, where the variable is located. The sytnax is:

```
<type>* <identifier>;
```

Or: `<type> *<identifier>;`

The memory address of a variable is determined using the **adress operator** "&":

```
&<identifier>
```

The variable the pointer points to is accessed using the **dereference operator** "*":

```
*<Pointer>
```

Pointer (Example)

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

int main()
{
  int i = 5, j = 10; // two int variables
  int* p; // a point to an int (uninitialised)

  p = &i; // p points to i

  cout << "p: " << p << endl;  // memory location, p points to
  cout << "*p:" << *p << endl; // value of the variable i

  p = &j; // p point to j

  cout << "p: " << p << endl;  // memory location, p points to
  cout << "*p:" << *p << endl; // value of the variable j
}
```

Null pointer

There exists a special pointer value 0 (NULL in C), pointing to the (hexadecimal) memory address 0x0 (= nil):

- It can be used for boolean tests of a pointer variable
- Used to detect memory issues
- If a pointer is not initialized with an address of a valid object, it should be initialized as a NULL-pointer
- Initialized with `<type>* <identifier> = 0;`
- Dereferencing the null pointer is undefined
- Note that a valid memory address, i.e., `!= nullptr` does not guarantee a valid value

C++11 Tip

The NULL-pointer can also be initialized by assigning the the keyword `nullptr`.

Null pointer

```
{
  double d = 1.345;
  double *p1 = 0, *p2 = &d;

  if (p1 == nullptr)
    std::cout << "p1 is invalid.";

  if (p1 != 0)  // Here, 0 is a memory address
                // Equivalent to nullptr or NULL
    std::cout << *p1; // p1 is not a valid pointer, undefined, do not
    use!

  if (p2)  // true if the memory address is not nullptr
    std::cout << *p2; // p2 ist a valid pointer, can be used
}
```

Pointer to constants and constant pointer

Pointer can be `const`. This means that the pointer (i.e. the memory location it points to) can not be modified. This must be distinguished from pointer to constant variables. Such pointer can be modified, but the underlying variable value is kept constant.

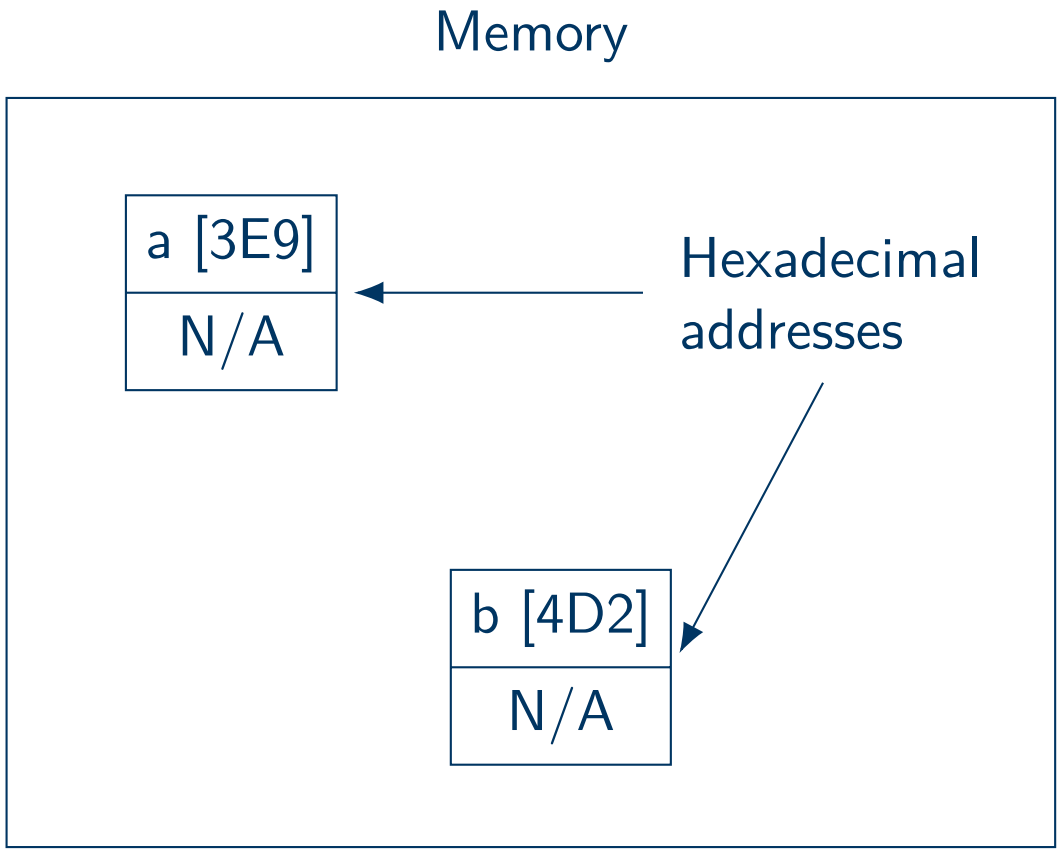
```
{
  const int a   = 1;
  int b         = 2;
  const int *p1 = &a; // ok, variable pointer, value not modifiable
  const int *p2 = &b; // ok, variable pointer, value not modifiable
  int const *p3 = &b; // same as const int *
  int *p3       = &a; // Error: int* pointing to const int

  int* const p4 = &b; // ok, constant pointer, value modifiable
  *p4           = 5;  // ok, pointer not modified, value modified
  p4            = &a; // Error: pointer can not be modified
}
```

Pointer arithmetic

Arithmetic operations can be applied to pointers:

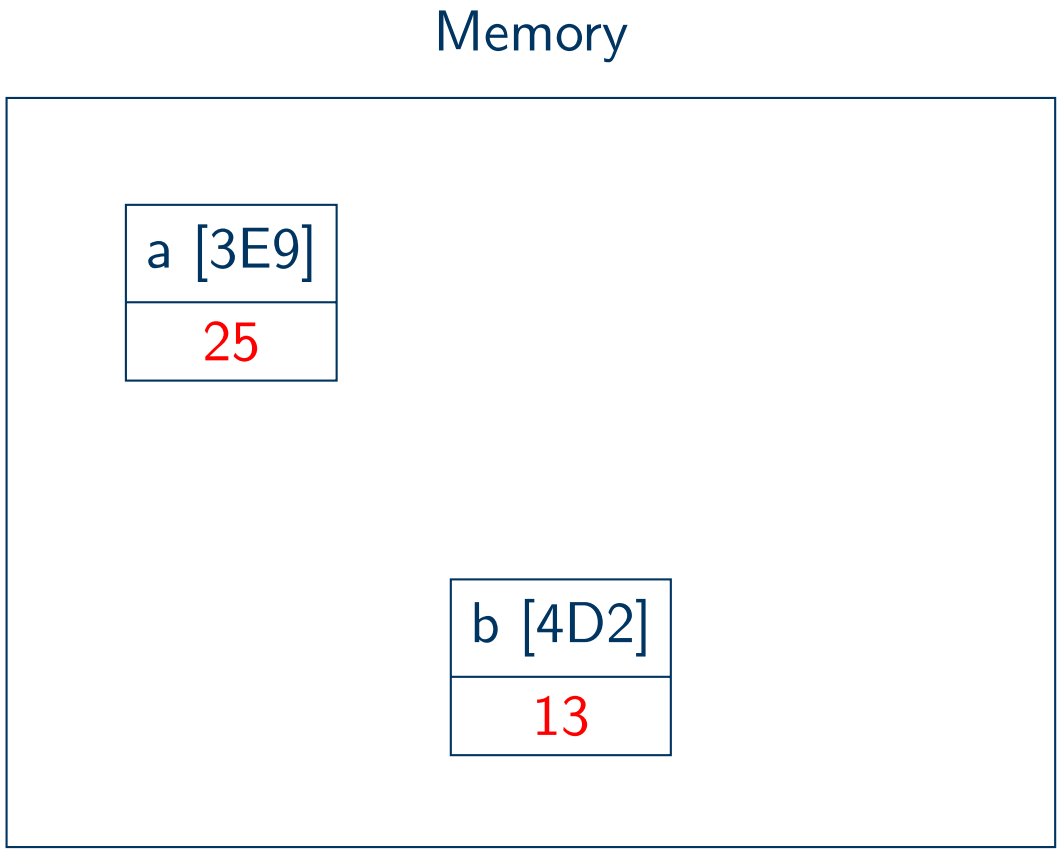
- `==, !=, <, >, <=, >=`: comparison w.r.t memory location
- `+ / -`: addition, subtraction (e.g. memory distance)
- `p + N` := `p + N*sizeof(<type>)`
- `++, --`: increment, decrement

Pointer - example



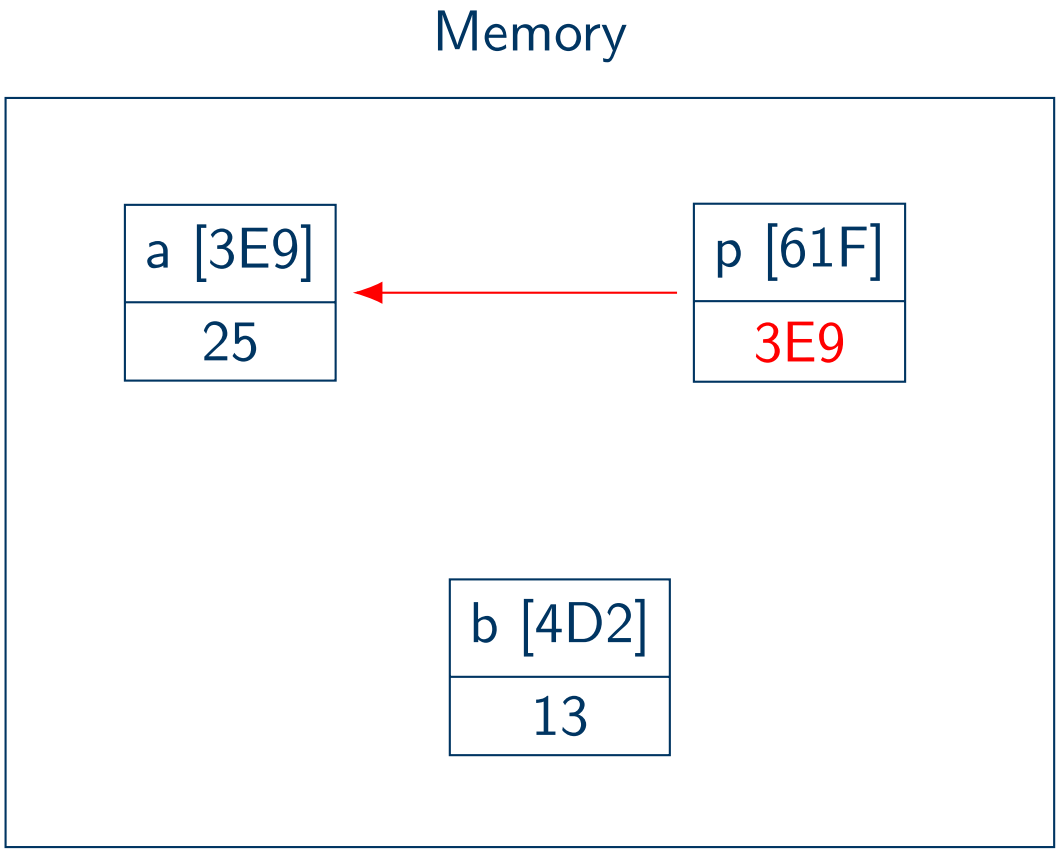

```
int a;
int b;
```

Pointer - example



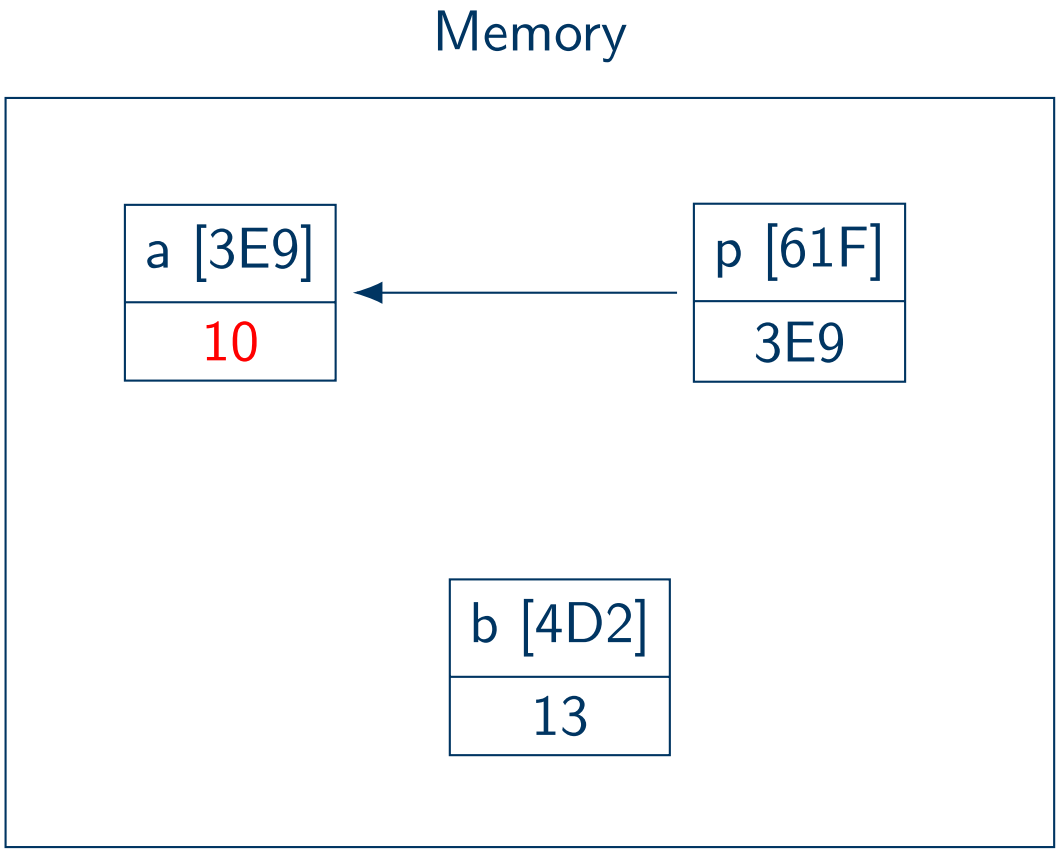

```
int a;
int b;
a = 25;
b = 13;
```

Pointer - example



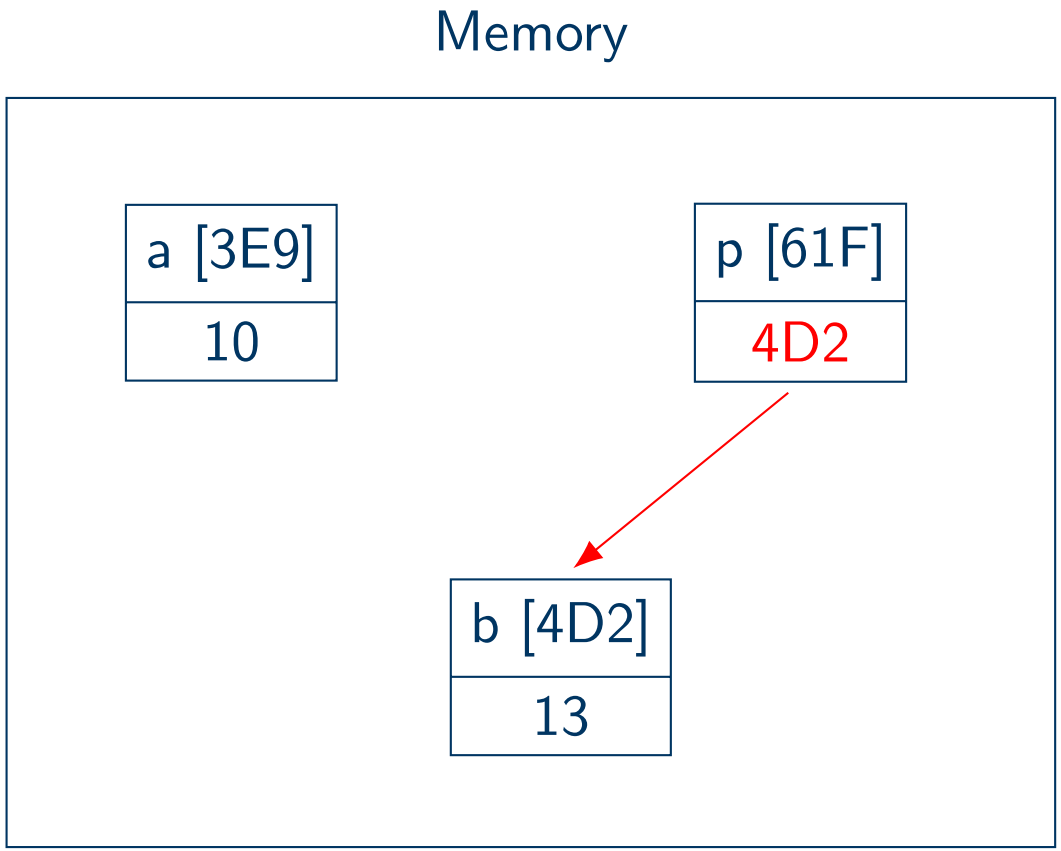

```
int a;
int b;
a = 25;
b = 13;
int *p;
p = &a;
```

Pointer - example



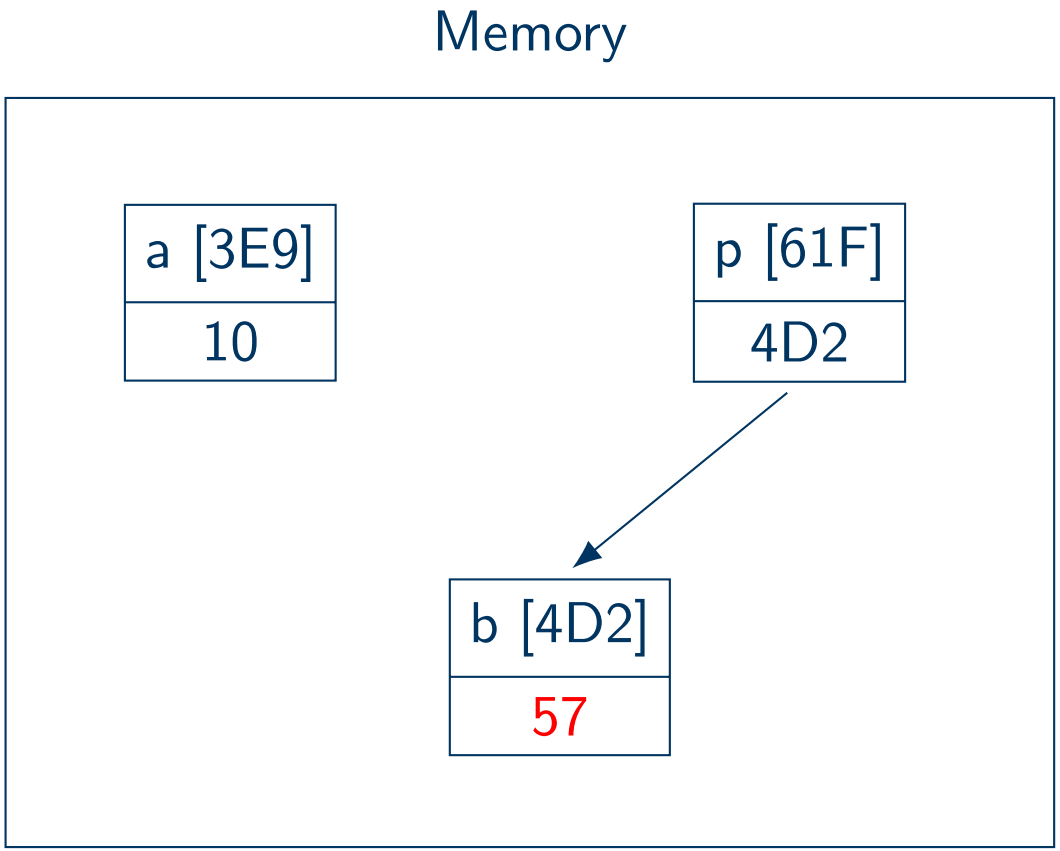

```
int a;
int b;
a = 25;
b = 13;
int *p;
p = &a;
*p = 10;
```

Pointer - example




```
int a;
int b;
a = 25;
b = 13;
int *p;
p = &a;
*p = 10;
p = &b;
```

Pointer - example




```
int a;
int b;
a = 25;
b = 13;
int *p;
p = &a;
*p = 10;
p = &b;
*p = 57;
```

Arrays and Pointer-Array Conversions

Array

An **array** groups data of the same type. It can be considered as a vector. It is declared with `<type> <identifier> [<size>];`. Importantly, `<size>` has to be known at compilation time.

```
int main(){
  int vMyArray[5]; // Array of size 5
  vMyArray[0] = 2; // Access first element
  vMyArray[4] = 3; // Access last element
}
```

An array of size N is accessed via indices $0, \ldots, N-1$.
The access operator is [ ].

Array Initialization

With C++11, there exist many ways to initialize an array:

```
int main(){
  int numbers[4] = {1, 4, 7, 9};
  double decimals[10];
  float prices[3] = {3.2}; // First element is 3.2, remaining 0
  int earnings[2] {7,9}; // Also allowed without =
  double wages[] = {35.7, 40.2, 42}; // Compiler counts elements
  float amounts[7] = {}; // All elements initialized with 0
}
```

The C++ Standard Template Library (STL) provides alternatives to arrays with the vector template class and array template class (in C++11). These will be briefly introduced in section 14.

Pointers and Arrays

An array uses consecutive memory, i.e. the next array element is physically located at the next memory location.

Pointer variables can be used as array identifiers and vice versa.

```
{
  int a[3] = {4,5,6};
  int* p;

  p = &a[0];          // p points to the start of the array a
  std::cout << p[1]; // returns the 2nd position value of the array

  std::cout << a[1] << p[1] << *(a+1) << *(p+1); // all the same
}
```

`(a+1)` references the memory address of a plus the memory amount of a single variable of a's type (e.g. 4 bytes for a int variable). Therefore, `*(a+1)` dereferences the same value as `a[1]`.

Multidimensional arrays

Multidimensional arrays can be constructed: More context?

```
{
  const int NumRows = 2;
  const int NumCols = 3;
  int a[NumRows][NumCols] = {{1,2,3},{4,5,6}};

  for(int i = 0; i < NumRows; ++i){
    for(int j = 0; j < NumCols; ++j){
      std::cout << a[i][j] << ' ';
    }
    std::cout << std::endl;
  }
}
```

Memory Management

Static Memory Usage

Static and Dynamic Memory

Up until now, memory was allocated during compilation of the program.

- This is known as **static allocation**.
- The size and location of the variables are fixed and they are located in the **stack**.
- As a consequence, the amount of memory required has to be known during compilation time.
- The occupied memory will be automatically released at the end of the respective scope.
- Dynamic allocation offers a more flexible approach to memory management.

Dynamic Allocation

Explicitly allocating memory

The memory for variables within a block is allocated automatically and released at the end of the block.

Memory can also be managed by the user. The memory can be allocated dynamically and **must** also be released by the user. (no garbage collection)

Allocate: `<Type>* p = new <Type>;`
Free: `delete p;`

Allocate: `<Type>* p = new <Type>[<Size>];`
Free: `delete[] p;`

Allocation (Example)

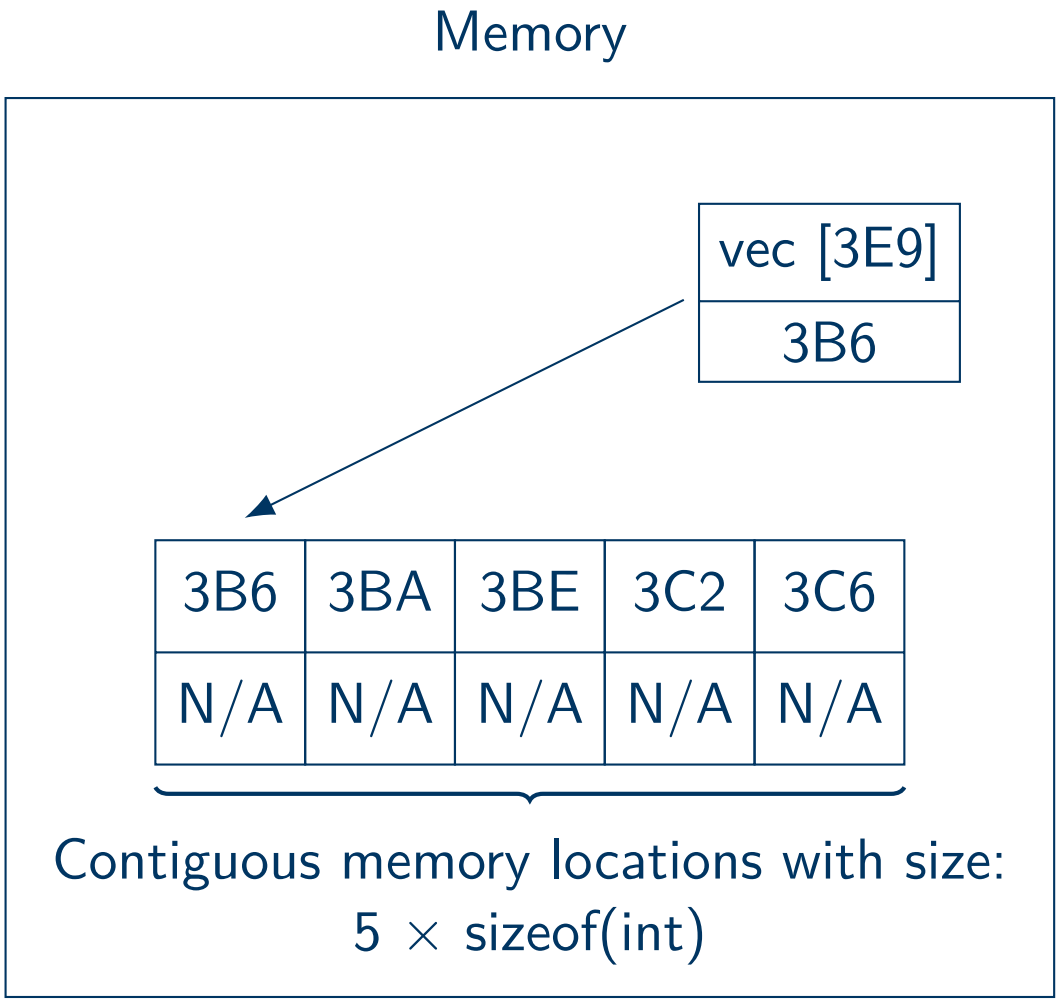
```
{
 int* a;
 {
   a = new int[10]; // allocate memory
 }
 a[2] = 5; // memory still valid

 delete[] a; // memory deallocated
}
```

Allocation (Attention)

```
{
 {
   int* a = new int[10]; // Memory allocated
 }

 // No pointer access anymore, but memory still allocated
}
```

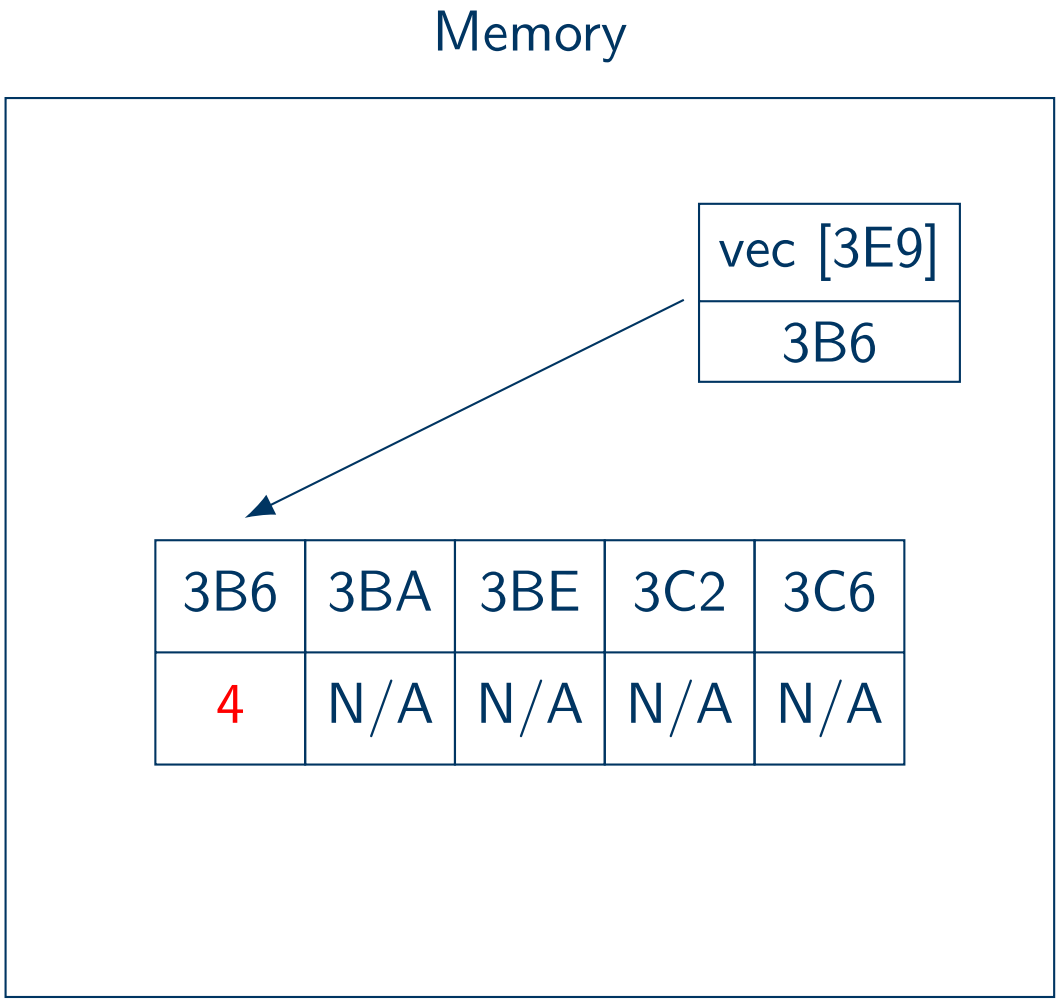
The **user must** free the dynamically allocated memory!

Dynamic allocation - example



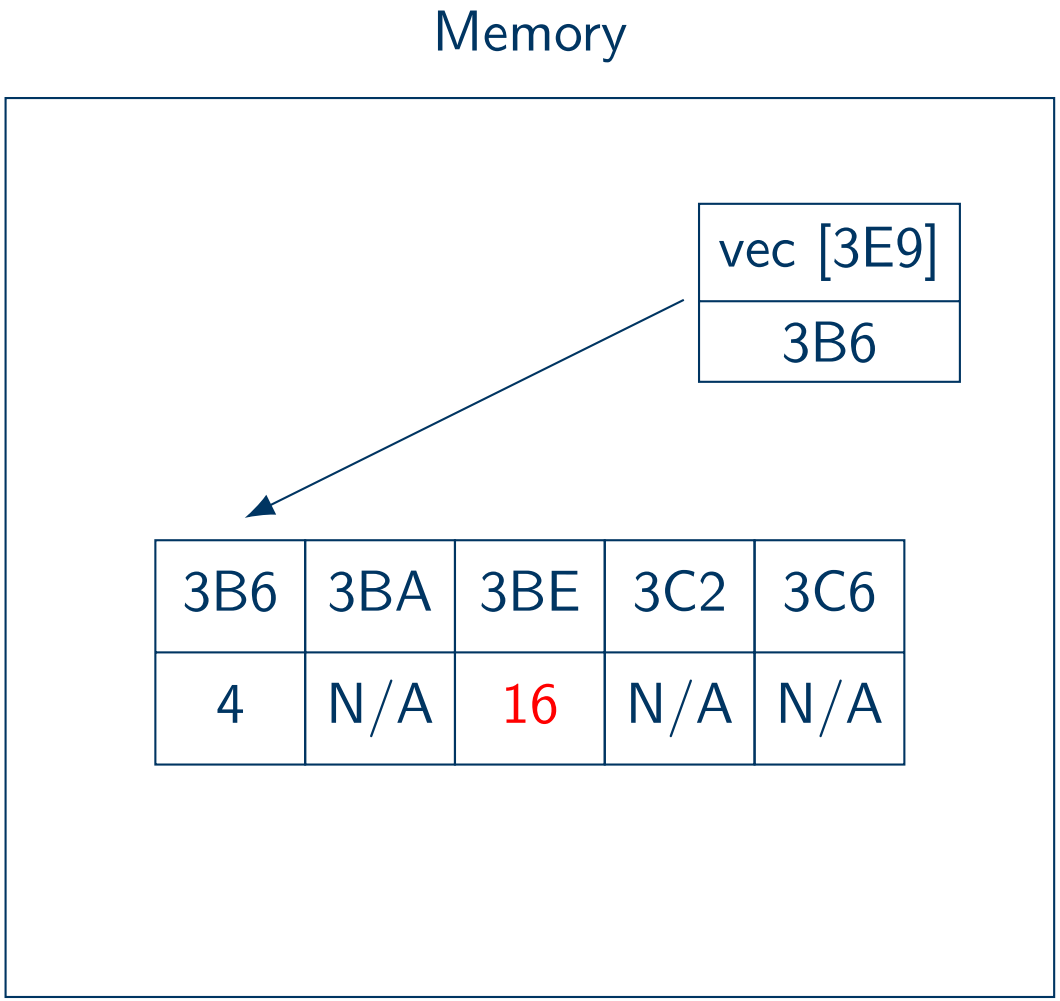

```
int * vec;
vec = new int[5];
```

Dynamic allocation - example



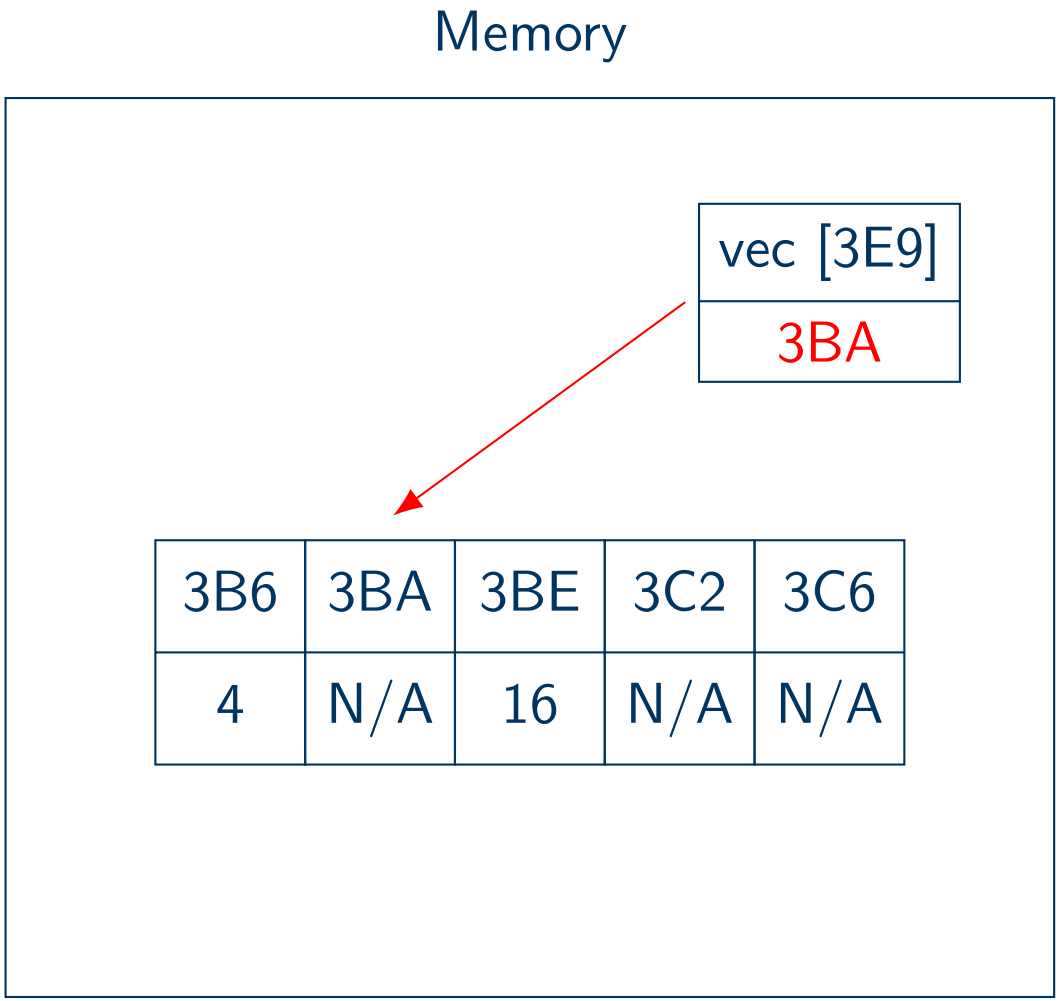

```
int * vec;
vec = new int[5];
vec[0] = 4;
```

Dynamic allocation - example



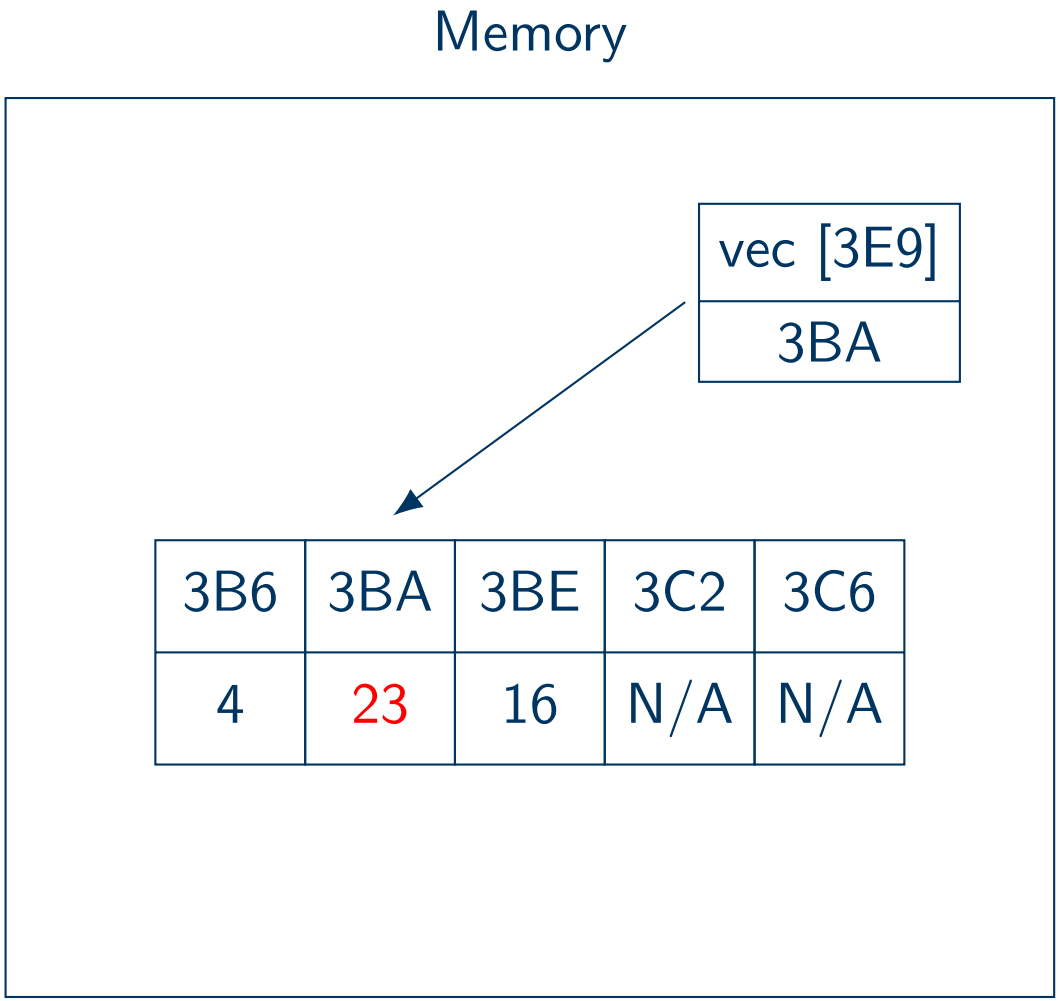

```
int * vec;
vec = new int[5];
vec[0] = 4;
*(vec + 2) = 16;
```

Dynamic allocation - example



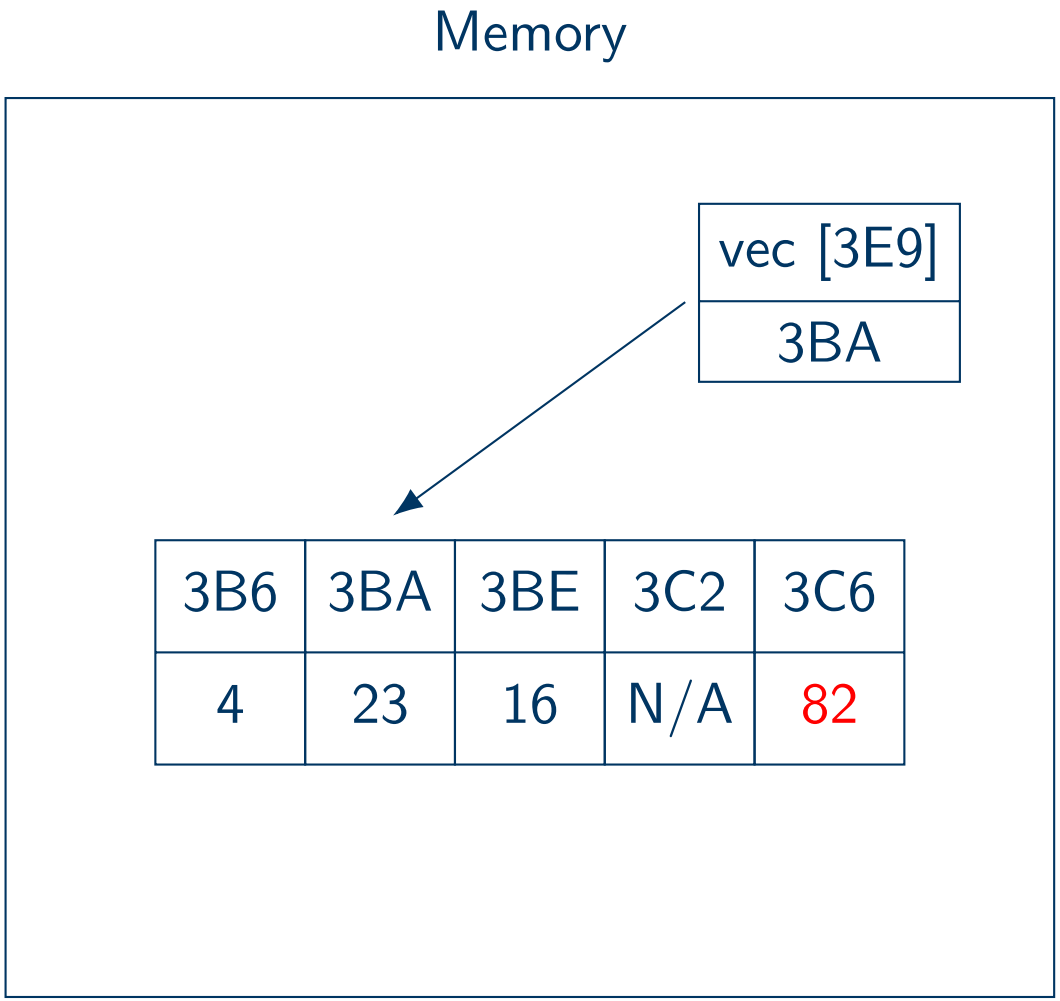

```
int * vec;
vec = new int[5];
vec[0] = 4;
*(vec + 2) = 16;
++vec;
```

Dynamic allocation - example




```
int * vec;
vec = new int[5];
vec[0] = 4;
*(vec + 2) = 16;
++vec;
vec[0] = 23;
```

Dynamic allocation - example




```
int * vec;
vec = new int[5];
vec[0] = 4;
*(vec + 2) = 16;
++vec;
vec[0] = 23;
*(vec + 3) = 82;
```

Functions

Functions

A function ...

- processes a closed subtask
- structures the program
- makes the code reusable
- allows to export the functionality (e.g. as a library)

Definition:

```
<ReturnType> <fctName> ( <paramList> ) {
   <statement>
}
```

Declaration/Definition of a function

The **signature** of a function are name and parameter list.

The **function header** are return type, name and parameter list.

The **function body** is the block with statements.

A function must be **defined** exactly once. The definition of a function consists of the function header and the function block.

A function can be **declared** several times. The declaration of a function consists only of the function header.

Definition of a function (Example)

```
// Declaration
int Add(int x, int y);

int main(){
  Add(3,4); // use function
}

// Definition
int Add(int x, int y){
 return x+y;
}
```

A function can be used whenever it has been declared. The definition can be stated later or be provided by a library in the linking process.

Return values

There exist two types of functions:

- Those that do not return a value or object, called `void` functions
- Those that return an object

The `<ReturnType>` in the beginning of the declaration defines the type of returned value. If an object or value is returned, the function has to finish with `return <object>`.

```
..
int Add(int x, int y){ // return an int
 return x+y; // x+y is an object of type int
}

void DisplaySum(int x, int y){ // function will not return anything
 std::cout << x+y << std::endl;
}
..
```

Function declaration

Why does a function need to be declared first?
A declaration describes the properties of the function to the compiler. With this information the compiler can efficiently check if a function call is valid without looking at the full definition. This is especially helpful when the program is spread over several files.

```
int Add(int x, int y); // Declaration

int main(){
  Add(3,'A'); // compiler detects error without
              // checking the full definition
}
```

The variable name can also be dropped in the declaration.

```
int Add(int, int); // Valid declaration
```

Function overloading

Several functions with the same name can coexist. In this case, the parameter list must differ (return value not sufficient). These functions are called **overloaded**.

```
int Add(int x, int y);

double Add(double x, double y);  // ok, different type
int Add(int x, int y, int z);    // ok, different number of params

double Add(int x, int y);        // Error: only different return type
```

Variables within functions

Variables within functions

The scope rules for variables in a function are the usual ones - a function behaves as a block.

Variables in the parameter list are local variables (i.e. only visible within the function and deleted at the end of the funtion).

Exception: **static** variables. A variable daclared as `static` is initalized on the first use and the value persists in succeeding calls.

```
void print(){
  static int val = 1;
  std::cout << "val: " << val++; // calls: 1, 2, 3, ...
}
```

Call by value

Except for pointers and references, a variable value is **copied** into the parameter (call by value) during a function call.

```
void AddOne(int x){
  x += 1;
  std::cout << "x: " << x; // Output: 6
}

int main(){
  int a = 5;
  AddOne(a);
  std::cout << "a: " << a; // Output: 5
}
```

The passed object is **not modified**.

Call by reference

For references in the parameter list, the **reference** to the variable is passed (call by reference) during a function call.

```
void AddOne(int& x){
  x += 1;
  std::cout << "x: " << x; // Output: 6
}

int main(){
  int a = 5;
  AddOne(a);
  std::cout << "a: " << a; // Output: 6 (!!)
}
```

The passed object is **modified**. Within the function it simply has a different name.

Call by pointer

For pointers in the parameter list, the **memory location** of the variable is passed.

```
void AddOne(int* x){
  *x += 1;
  std::cout << "x: " << *x; // Output: 6
}

int main(){
  int a = 5;
  AddOne(&a);
  std::cout << "a: " << a; // Output: 6 (!!)
}
```

The passed object is **modified**. Within the function it simply has a different name (analogously to: call by reference).

Functions and Arrays

Passing arrays

Arrays can be passed as parameters.

```
void AddOne(int x[]){
  x[0] += 1;
  std::cout << "x: " << x[0]; // Output: 6
}

int main(){
  int a[20]; a[0] = 5
  AddOne(a);
  std::cout << "a: " << a[0]; // Output: 6 (!!)
}
```

Passing multidimensional arrays

Multidimensional arrays can be passed to a function.
Pointer to multidimensional arrays must specify all but the first size explicitly.

```
void SomeFct(int (*x)[100]){
  x[0][0] += 1;
  std::cout << "x: " << x[0][0]; // Output: 6
}

int main(){
  int a[20][100];
  int (*pArray)[100] = a; // matching pointer
  SomeFct(pArray);
}
```

Variable multidimensional arrays

If the size of a two-dimensional array is not known at compile time, it is beneficial to proceed as follows:
Every matrix row is stored in a 1d-array that is allocated via `new`. These matrix rows are grouped in an array of pointers — again allocated via `new`.

```
{
  int z = 3, s = 4;             // non-constant values
  int** mat = new int*[z];      // an array of int* (rows)
  for(int i = 0; i < z; ++i)    // for each row
    mat[i] = new int[s];        // allocate memory for the row

  std::cout << mat[2][3];       // access
}
```

Attention: call by reference

On passing parameter by reference or by pointer, one must be aware that the values of the variables can be modified.

If one wants to avoid a copy (e.g. for large objects), but to ensure not to modify the variable, the object should be passed as a **`const` reference**. The compiler will check if this requirement is matched. The caller of the function can thus be sure that the object is not modified.

```
void AddOne(const int& x){
  x += 1; // Error: can not be modified
}

int main(){
  int a = 5
  AddOne(a);
}
```

Default parameter

Parameters of a function can be declared with default values. These values are used if the parameters are omitted in the function call.

```
bool IsSmall(double x, double tol = 1e-3){
  if(x < tol) return true; else return false;
}

int main(){
  std::cout << IsSmall(1e-15); // true
  std::cout << IsSmall(1e-15, 1e-18); // false
}
```

Return values

The return value of a function can be passed by value or by reference. Returning by reference must ensure, that the variable still exists after the function is completed. **Local variables must not be returned by reference!**

```
int& Increment_Good(int& x){
  return ++x;
}

int& Increment_Bad(int x){
  return ++x;
}

int main(){
  int a = 1;
  std::cout << Increment_Good(a); // ok
  std::cout << Increment_Bad(a);  // Error: return value (x) does not
    exist anymore
}
```

`inline`-Functions

A function call takes time: parameter must be copied, the program jumps to another location and after completion the return value has to be treated. This effort should be avoided for small functions.

Therefore, a function can be declared `inline`. This suggest to the compiler to insert the function inplace instead of the function call. It is just a suggestion to the compiler, but usually done.

```
inline int Add(int x, int y) { return x+y;}

int main(){
  std::cout << Add(3,4);  // inline function call
}
```

Pointer and functions

Functions are located somewhere in the memory. Therefore, one can also store a pointer to the function location.

```
int max(int a, int b) {if(a>b) return a; else return b;}
int min(int a, int b) {if(a<b) return a; else return b;}

{
  int (*pF)(int,int);  // pF is a pointer to a function

  pF = &max;
  std::cout << (*pF)(3,4); // returns maximum

  pF = &min;
  std::cout << (*pF)(3,4); // returns minimum
}
```

Const correctness

Const correctness

`const` correctness describes the usage of the keyword `const` to prevent objects from being mutated. It can be applied to variables, pointers, functions and more complicated objects such as classes. Using `const` is a good programming practice and provides several benefits:

- It protects the user from accidentally changing a variable that is not intended to be changed

```
const int a = 2;
a = 19; // error, a is read-only
```

- It protects the user from simple typing errors, for example assignments:

```
const int a = 2;
if (a = 1) {...} // user typing error, compiler will detect
```

- The compiler can optimize the code more efficiently

Pointer to constants and constant pointer

Pointer can be `const`. This means that the pointer (i.e. the memory location it points to) can not be modified. This must be distinguished from pointer to constant variables. Such pointer can be modified, but the underlying variable value is kept constant.

```
{
  const int a = 1;
  int b = 2;
  const int* p1 = &a; // ok, variable pointer, value not modifiable
  const int* p2 = &b; // ok, variable pointer, value modifiable
  int* p3 = &a;       // Error: int* pointing to const int

  int* const p4 = &b;    // ok, constant pointer, value modifiable
  *p4 = 5;               // ok, pointer not modified, value modified
  p4 = &a;               // Error: pointer can not be modified
}
```

Const correctness and functions

Ensuring that an object will not be modified when passed to a function can be done in multiple ways.

```
..
void f1(const int& a); // call by const reference
void f2(const int* b); // call by const pointer
void f3(int c); // call by value
..
```

If `f1` or `f2` attempt to change the passed integer, the compiler will flag this as an error at compile-time. On the other hand, `f3` cannot even change the integer, since it is a call by value. A local copy of `c` will be created that is destroyed when `f3` returns.

Modular Program Design

Modular program design

Large programs raise the question how to split the program into meaningful subcomponents.

In general, independent functionality should be split into **different files**. However, similar functionality can be grouped in a single file.

Declaration should be separated from the definition. Usually, the implementation is located in a *.cpp file, a declaration is found in a *.h file.

This way, other parts of the program only need to include the header (*.h), i.e. only the declaration.

Example

```
int Add(int x, int y);
```

add.h

```
#include "add.h"
int Add(int x, int y){
    return x+y;
}
```

add.cpp

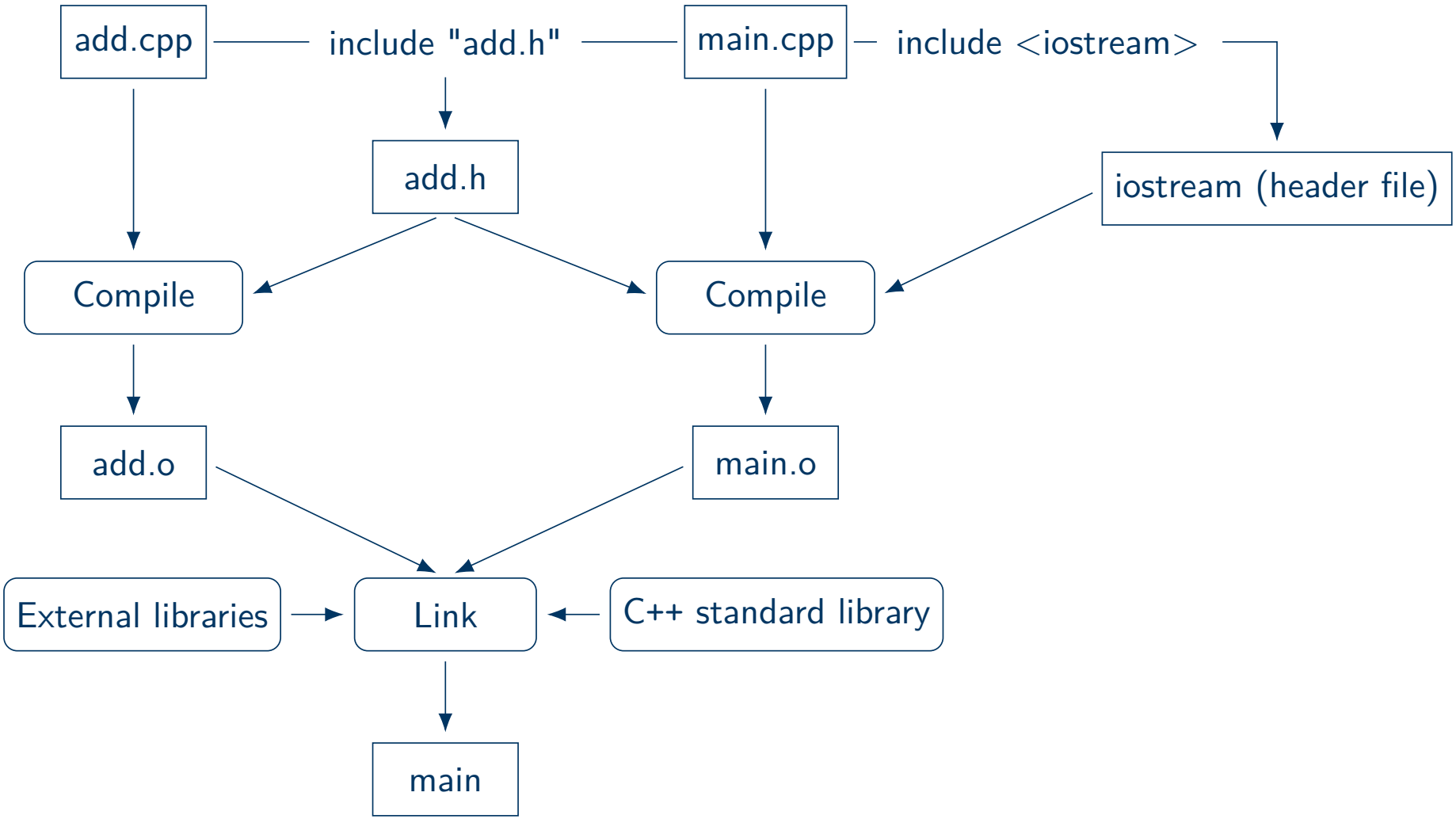
```
#include "add.h"
int main(){
    std::cout << Add(5,6);
}
```

myProc.cpp

Advantage:

- program modular stuctured
- program parts can be compiled independently
- implementations can be replaced
- compile time is reduced

Example


add.cpp
include "add.h"
main.cpp
include <iostream>
add.h
iostream (header file)
Compile
Compile
add.o
main.o
External libraries
Link
C++ standard library
main

Include guards

Files typically include several *.h files and headers may include other headers. To ensure that declarations are not compiled to often, so called **include guards** are used. They ensure that a header content is included only once.

```
#ifndef __H__SOME_HEADER__
#define __H__SOME_HEADER__

// .. declarations here

#endif // end __H__SOME_HEADER__
```

someHeader.h

The name of the include guards must be unique.

Directory management

Static and dynamic linking

When **static linking** is used, the used library code is directly copied into the final binary file during the linking process. Therefore, the binary works as a standalone file. If the linked library or the source file is changed, the user has to compile and link again.

With **dynamic linking**, the dependencies are built during runtime. Therefore, the library file has to be present during execution and no recompilation is necessary if the library is changed.

Static linking results in an larger executable file, but distribution is a lot easier. On the other hand, maintenance requires more work compared to dynamic linking, as changes in a library will not be automatically incorporated.

User-defined and Composite Data Types

Structures - motivation

C, the language on which C++ was built, is a **procedural** language, which emphasizes the use of functions (procedures) to structure and modularize a program.

C++ is an **object-oriented** programming (OOP) language, where objects are used to break down a programming task. These objects contain both data and functions.

In C++, **classes** are mainly used to enforce this programming paradigm. The advanced course will discuss classes in detail. An important stepping stone towards classes are **structures**, which declare an user-defined composite type consisting of different types.

Structures - example

Suppose you need to work with complex numbers. Each number is split into a real and a complex part. Ideally, a data type exists that stores all values in one unit. As soon as we use different data types, an array will not work, since it can only hold values of the same type.
A C++ structure, also called **struct**, is the answer. All related informations for one complex number can be stored in a single struct variable.

```
struct <name> {
     <type1> name1;
     <type2> name2;
     ...
};
```

Structures - example

A simple struct that models a complex number:

```
struct Complex { // Structure declaration
  float re; // Real
  double im; // Imaginary
  bool isComplex;
}; // Semicolon to finish the declaration!
```

Two common initializations can be used: the {} notation, where the elements are defined sequentially and separated by a comma.

```
Complex x = {2.2, 1.7, true};
```

Or with the membership . (dot) operator, which can access each element individually.

```
Complex y;
y.re = 4.5; y.im = 6.8; y.isComplex = true;
```

Structures - example

```
#include <iostream>
struct Complex { // Structure declaration
  float re; // Real
  double im; // Imaginary
  bool isComplex;
}; // Semicolon to finish the declaration!
int main(){
  Complex x = {
    2.2,
    1.7,
    true
  };
  Complex y;
  y.re = 4.5; // Using the membership operator
  y.im = 6.8;
  y.isComplex = true;
  std::cout << "x= " << x.re << " + " << x.im << " i" << std::endl;
}
```

Other initializations

C++11 initialization

The = is optional:

```
Complex x {2.2, 1.7, true}; // x.re=2.2, x.im=1.7, x.isComplex=true
```

Empty braces result in the individual members being set to their default values.

```
Complex z {}; // z.re=0, z.im=0, z.isComplex=0/false
```

One can also create an array of structures, for example to store several complex values.

```
Complex point2d[2] = {
  {6.7, 9.12, true},
  {2.3, 1.4, true}
};
```

Structures with dynamic memory allocation

For efficient memory usage, it is often beneficial to create structures during runtime. This will allocate the memory dynamically. A pointer and the `new` operator is used for initialization:

```
Complex * z = new Complex;
```

To access elements with a pointer, the `->` operator is used:

```
z->re = 12.78; z->im = 3.9; z->isComplex = true;
```

Since `z` is a pointer to a structure, `(*z)` is the structure itself. Therefore, the elements can also be accessed with the dot operator:

```
(*z).re = 12.78; (*z).im = 3.9; (*z).isComplex = true;
```

Finally, the memory has to be deallocated by the user:

```
delete z;
```

Structures and functions

Unlike arrays, structures are treated similar to a basic type when passed to a function.

```
void Show(Complex x){
  if (x.isComplex == True){
    std::cout << "Real part: " << x.re << ", Complex part: " << x.im;
  } else {
    std::cout << "Number is not complex with real part: " << x.re;
  }
}
```

Dependent on the size of the structure, calling by value is not efficient since a copy has to be created. Passing the memory location is a better option for larger structures.

Call by pointer - example

```
#include <iostream>
struct Complex {
  float re;
  double im;
  bool isComplex;
};
Complex add(Complex, Complex); // Declaration
int main(){
  Complex a {1.2, 7.4, true};
  Complex b {5.4, 2.3, true};
  Complex z = add(&a, &b);
}

Complex add(Complex* x, Complex* y){
  Complex z;
  z.re = x->re + y->re;
  z.im = x->im + z->im;
  return z;
}
```

Call by pointer - example

```
#include <iostream>
struct Data {
 int arr[10000];
};
Data add(Data, Data); // Declarations
Data add(Data*, Data*); // Call by pointer and function overloading
Data add_ref(Data&, Data&);  // Declaration using call by reference
int main(){
  Data a {{1,2,3,...}};
  Data b {{99,98,97,...}};
  Data u = add(a, b); // Inefficient call by value
  Data v = add(&a, &b); // More efficient call by pointer
  Data w = add_ref(a, b); // More efficient call by reference
}
inline Data add(Data x, Data y){ someCode, ...; return someData; }
inline Data add(Data* x, Data* y){ someCode, ...; return someData; }
inline Data add_ref(Data& x, Data& y){ someCode, ...; return someData;}
```

Enumerations

Enumerations are a user-defined data type consisting of named values that represent integral constants. Enumerations have a fixed range of possible values.

`enum Typname { <name1> [=<value1>], <name2> ...};`

```
enum Season {spring, summer, autumn, winter};
int main(){
  Season feb = spring;
  std::cout << feb << std::endl; // Output: 0
  feb = winter;
  std::cout << feb << std::endl; // Output: 3
}
```

Enumerations

Enumerations are user-defined counting types. They serve as a human readable numbering.

`enum [class] Typname { <name1> [=<value2>], <name2> ...};`

```
enum Color {red = 0, blue, yellow};
int main(){
  Color x = red;
  x = blue;
}
```

Union

A **Union** is a data format that can hold different data types but only one type at a time. All components of a union are stored in the same memory location.

```
union <name> {
      <type1> name1;
      <type2> name2;
      ...
};
```

```
union Number {
  int i;
  float f;
};
int main(){
  Number x;
  x.i = 123; // x stores an int
  x.f = 1.7; // Stores a double, int is lost
}
```

Union

All components of a **union** are stored in the *same* memory location.

```
union <name> {
      <type1> name1;
      <type2> name2;
      ...
};
```

```
union Number {
  int i;
  float f;
  double d;
};
int main(){
  Number x;
  x.i = 123; x.f = 1.7; x.d = 2.3;
}
```

typedef

Using `typedef`, an alias of a type can be declared.

`typedef <type> <typAlias>;`

```
typedef int MyInt;
typedef float Point2d[2];
typedef struct {double x,y,z;} Point3d;
int main(){
  MyInt a = 4;    // a is a int
  Point2d p2d:    // p2d is a float-array of size 2
  Point3d p3d;
}
```

If a type has to be changed afterwards, for example from `int` to `long int` to support larger integers, `typedef` simplifies the process considerably, as the type has to be changed only once.

File Input/Output

File output

To interact with files, the file input/output class `fstream` from the standard library is used. This library combines `ofstream`, used for file output, and `ifstream`, used for file input, into one library (similar to `iostream`, which combines console in- and output).

For **file output**, there are a couple of simple steps:

- Include the `fstream` header file
- Create an `ofstream` object
- Associate the `ofstream` object with a file
- Use the `ofstream` object in the same manner you would use `cout`
- Close the file

File output - example

```cpp
#include <iostream>
#include <fstream> // Include the file input/output library

int main () {
  std::ofstream myfile; // Create ofstream object
  myfile.open("example.txt"); // Associate file
  myfile << "Writing this to a file." << std::endl;
  myfile << "New line, 2 + 2 = " << 2+2 << std::endl;
  myfile.close(); // Close the file
}
```

Caution: Once the file is associated, the previous contents will be erased! To append at the end of the existing content, the following flag is used:

```cpp
myFile.open("example.txt", std::ios::app);
```

File input

Interacting with a file as input requires data, that can hold the information from the file. For character-based input, we will use the basic type `char` and `strings` from the C++ standard library. A `string` represents a sequence of characters.

```
string text = "Hello";
```

Steps to read and save the content from a file:

- Include the `fstream` and `string` header files
- Create an `ifstream` object and associate it with a file
- Working with the file is possible with characters or strings:
  - With characters, read each character one by one using the `get()` function
  - With strings, read the content line-by-line with the `std::getline()` function
- Close the file

File input

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

int main () {
  ifstream myfile ("example.txt");
  char ch;
  if (myfile.is_open()){
    while(myfile.get(ch)){ // myfile.get(ch) saves the next character
                           // in ch and is also used as a statement
      cout << ch;
    }
    myfile.close();
  }
  else cout << "Unable to open file" << endl;
}
```

File input

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

int main () {
  string line;
  ifstream myfile ("example.txt"); // Creates ifstream object
                                   // and associates
  if (myfile.is_open()){
    while (getline(myfile,line)){
      cout << line << endl;
    }
    myfile.close();
  }
  else cout << "Unable to open file" << endl;
}
```

File input - strings and characters

In both examples, `myfile.open()` is used. It is a `bool` function from the STL that checks if a file is associated to an `ifstream` object.

If the input is processed using characters, the function `myfile.get(ch)` is used. It will assign the next character (including whitespaces) to the passed `char` object.

The function `std::getline(istream&, string&)` saves the next line from the `istream` object into the passed `string`.

Both functions can be used in conditional statements, where the return object will be converted into `true`, if either a string or character is returned, and `false`, if no character or string is left.

Error handling

Error handling

Errors are treated in C++ using **exceptions**. If an error occurs, an error object is **thrown**. This object can be **caught** at a different program location, in order to cope with the error. If the error is not caught, the program will terminate reporting the thrown error.

```
#include <iostream>

int main(){
  try{
    int* ar = new int[10000000000000];
  }
  catch (const std::bad_alloc& e){
    std::cout << "Oh out of memory...?" << std::endl;
  }
}
```

Overview on C++ exceptions

The exception `std::bad_alloc&` is part of a larger hierarchy of exceptions.

exception	Standard exception
bad_alloc	Thrown by new on allocation failure
bad_function_call	Thrown by empty function objects
logic_error	Error related to the internal logic of the program
runtime_error	Error that can only be detected during runtime

There exist further, more specific error classes that are attached to the logic and runtime errors. The concept of error classes will be further investigated in the advanced course.

Custom errors

Using `std::runtime_error(string&)` a custom runtime error exception can be declared.

```
double divide(double a, double b);

int main(){
  double x = divide(5,0);
}

double divide(double a, double b){
  if (b==0) throw std::runtime_error("divide by zero");
  return a/b;
}
```

Custom error classes

Custom error class can be declared.

```
class DivideError{
   public:
      DivideError(double a, double b) : x(a), y(b) {}
      double x, y;
};

double Divide(double x, double y){
  if( y == 0 ) throw DivideError(x,y); // Cannot divide by zero
  return x / y;
}

int main(){
   try{
     Divide(5, 0);
   } catch( DivideError& ex ) {
     cout << "Error: Division "<<ex.x<<" / "<<ex.y<<endl;
   }
}
```

Debugging using assert

In order to check the validity of values at certain program points, one can use assert. An assertion is only ensured in debug modus. Thus, it serves to detect errors during the code development phase. The check is skipped in release mode. Then, the program has a higher performance.

```
#include <cassert>

int Divide(int x, int y){
   assert(y != 0);     // assert nonzero divisor
   return x/y;         // now ensured to be well-defined
}
```

The C++ Standard Library

C++ standard library

The usability of a programming language is often not only due to its own native concepts, but also to the libraries available. If you want to write a program, you want to code as little as possible yourself and reuse many of the existing libraries.

The C++ standard library provides many concepts and algorithms that can be used for one's purposes. Large parts of the library are implemented via templates to allow for adapted code with high performance.

The template part of the library is called **STL (Standard Template Library)**.

Content of the standard library

The standard library contains

- containers and iterators (vector, list, ...)
- standard algorithms (sort, search, ...)
- strings
- input/output (I/O)
- error handling (exceptions, assert, ...)
- mathematical functions (fabs, exp, sin, ...)
- ...

Complete references can be found, e.g., at:
`http://www.cplusplus.com`
`http://en.cppreference.com/`

Container, iterators and algorithms

Containers, Iterators, Algorithms

Container
A container is an object containing other objects.
(e.g.: lists, vectors, sets, . . .)

Iterators
Iterators allow to loop over the elements in a container.

Algorithms
Algorithms can be applied to a container, using the iterators to access the elements.

Overview container

A **container** is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows a great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides **member functions** to access them, either directly or through **iterators** (reference objects with similar properties to pointers).

	Array	Queue	Stack	Linked List	BST	Hash Table
Strength	Random access	FIFO	LIFO	Easy insert	Efficient search	Key-value map
Weakness	Static	No random access	No random access	No random access	Operations not always logarith-mic	Hash collisions
C++ STL	Vector/ Array	Queue	Stack	List	Set	Un-ordered set

Iterators

The C++ standard library offers so-called **iterators**, that are specifically designed to iterate through containers. An iterator points to a specific element of the corresponding container and is used similar to a pointer. It is initialized with

```
std::<containerType>::iterator <name>;
```

Each of the here presented containers provides a set of functions that can be used with an iterator:

begin()	Returns an iterator to the first element in the container
end()	Returns an iterator to the past-the-end element in the container Cannot be dereferenced, used to loop through a container
cbegin()	Returns a const iterator to the first element in the container The content it points to cannot be modified
cend()	Returns a const iterator to the past-the-end element in the container

Vector

A **vector** is a sequence container representing an array that can change in size dynamically. It is initialized with

`std::vector<type> <name>;`

```
#include <vector>

vector<int> vect1 = {2,6,9}; // = is optional
vector<double> vect2 (10);   // Initialized (empty) vector of size 10
vector<bool> vect3 (4,1);    // Four bool variables with value 1/true
```

Container offer a variety of **member functions**, used for easy handling.

assign(amount, value)	Assigns the new content and replaces the old values and size
push_back(value)	Adds a new element at the end of the vector
size()	Returns the size
empty()	Returns `true` if vector is empty and `false` otherwise

Vector - example

```cpp
#include <iostream>
#include <vector>
int main(){
  std::vector<int> vect;
  std::vector<int>::iterator it; // Initialize iterator
  vect.assign(5,11); // Assigns 5 ints with a value of 11
  vect[2] = 9;       // Access and change of third element to 9
  vect.push_back(4); // Appends a new integer with value 4
  // Output using vect.size():
  for (int i = 0; i < vect.size(); ++i)
    std::cout << vect[i] << std::endl; // Output: 11 11 9 11 11 4

  // Adding one to each element using iterator:
  for (it=vect.begin(); it != vect.end(); ++it)
    *it += 1; // Iterator is dereferenced
}
```

String

A **string** is an object that represents a sequence of characters. They are initialized with

`std::string <name>;`

```
#include <string>

std::string s1; // Empty string, length of 0 characters
std::string s2 ("A character sequence");
std::string s3 (s2); // s3 is a copy of s2
```

Some important **member functions** are:

size()	Returns the size of the string in bytes
empty()	Returns `true` if string is empty and `false` otherwise
clear()	Resets `string` to empty state
append(sting& str)	Appends `string str` to existing string

String - example

```
#include <iostream>
#include <string>
int main(){
  std::string str1 ("Hello");
  std::string str2 ("World");
  std::string str3 = str1 + " " + str2; // Concatenation
  std::cout << "Size of str3: " << str3.size() << std::endl;

  while(!str1.empty()){
    std::cout << str1.back(); // Returns last character of string
    str1.pop_back(); // Removes the last element
  }
  std::cout << std::endl;
  // Output using iterator
  for (std::string::iterator it = str3.begin(); it!=str3.end(); ++it)
    std::cout << *it; // Output: Hello World
}
```

Map

Maps are associative (non-linear) containers that store elements formed by a combination of a key value and a mapped value. Maps do not offer random access and are initialized with

`std::map <keyType, valueType> <name>;`

```
#include <map>

std::map <char, int> mymap; // Maps char keys to integer variables
```

Some important **member functions** are:

size()	Returns the number of elements in the map container
empty()	Returns `true` if the maps is empty and `false` otherwise
erase(key)	Removes the element associated with the key
clear()	Removes all elements from the map
count(key)	Counts the amount of times the key appears in the map
find(key)	Searches for the element and if found, returns an iterator to it

Map - example

```
#include <iostream>
#include <map>
int main(){
  std::map<char,int> mymap; // Initialize map
  std::map<char,int>::iterator it; // Initialize iterator
  mymap['a']=15; // Add key-value pairs to the map
  mymap['c']=7;
  mymap['e']=32;
  std::cout << "mymap now contains " << mymap.size() << " elements.\n";
  it = mymap.find('c'); // If the key does not exist, the
                        // iterator will point to mymap.end()
  if (it != mymap.end()) // Check if 'c' was found
    mymap.erase (it);    // Erase 'c' entry
  // Output using iterator:
  for (it=mymap.begin(); it!=mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
}
```

Unordered set

Unordered sets store elements in no particular order, which allows for fast retrieval of individual elements based on their value. They do not offer random access and are initialized with

`std::unordered_set<type> <name>;`

```
#include <unordered_set>

std::unordered_set<double> set1; // empty
std::unordered_set<int> set2 ({1,4,3,7}); // init list
```

Some important member functions are:

size()	Returns the number of elements in the unordered set
empty()	Returns `true` if the set is empty and `false` otherwise
insert(key)	Inserts a new element in the unordered set
erase(key)	Removes the specified element
find(key)	Searches for the element and if found, returns an iterator to it

Unordered set - example

```
#include <iostream>
#include <unordered_set>
#include <string>
using namespace std;
int main(){
  unordered_set<string> fruits ({"Apple", "Orange", "Peach"});
  string name;
  cout << "Enter fruit name: " << endl;
  getline(cin, name);
  fruits.insert(name);
  cout << "Size of fruits: " << fruits.size() << endl;

  // Output using const(!) iterator
  unordered_set<string>::const_iterator it;
  std::cout << "fruits contains:";
  for ( it = fruits.cbegin(); it != fruits.cend(); ++it )
    cout << " " << *it;    // cannot modify *it
  cout << endl;
}
```

Algorithms

The standard library also offers a wide range of **algorithms** that can be applied to a container. These algorithms will act on a range of elements, which are accessed through iterators. Calling an algorithm usually has the following form

```
std::<algorithm>(<beginIterator>, <endIterator>,
<additionalParameters1>, ...);
```

Some examples:

for_each(begin, end, function)	Applies the passed function to all elements
search(begin, end, sequence)	Searches range for subsequence
sort(begin, end)	Sorts container in ascending order
min_element(begin, end)	Returns iterator to smallest element in range
find(begin, end, element)	Returns iterator to element if present

And many more exist...

`http://www.cplusplus.com/reference/algorithm/`

Algorithms - example

```
#include <iostream>
#include <algorithm>
#include <vector>
void add (int& i) {i++;}
int main(){
  std::vector<int> vect;
  for(int i=4; i>=0; --i)
    vect.push_back(i*10);  // vect= 40 30 20 10 0

  std::for_each (vect.begin(), vect.end(), add); // Adds 1
  std::sort (vect.begin(), vect.begin()+3); // Only sorts the
                                            // first three elements
  auto it = min_element(vect.begin(), vect.end()); // Auto assign
  std::cout << "Smallest element in vect: " << *it << std::endl;
  std::cout << "Elements of vect: " << std::endl;
  for (it=vect.begin(); it != vect.end(); ++it)
    std::cout << *it << ' '; // Output: 21 31 41 11 1
}
```

Design principle for containers and algorithms

An algorithm can often be used without the exact knowledge of the underlying memory structure of a container. (e.g., searching for an element in a container)

In order to implement the algorithm only once, iterators are used. The iterator concept abstracts the access to elements of a container. This allows for generic implementation of the algorithms.

An array is a container. Pointers can be used to access an array. Therefore, the containers and iterators are designed such that also arrays and pointers satisfy the design principle.

Data structures

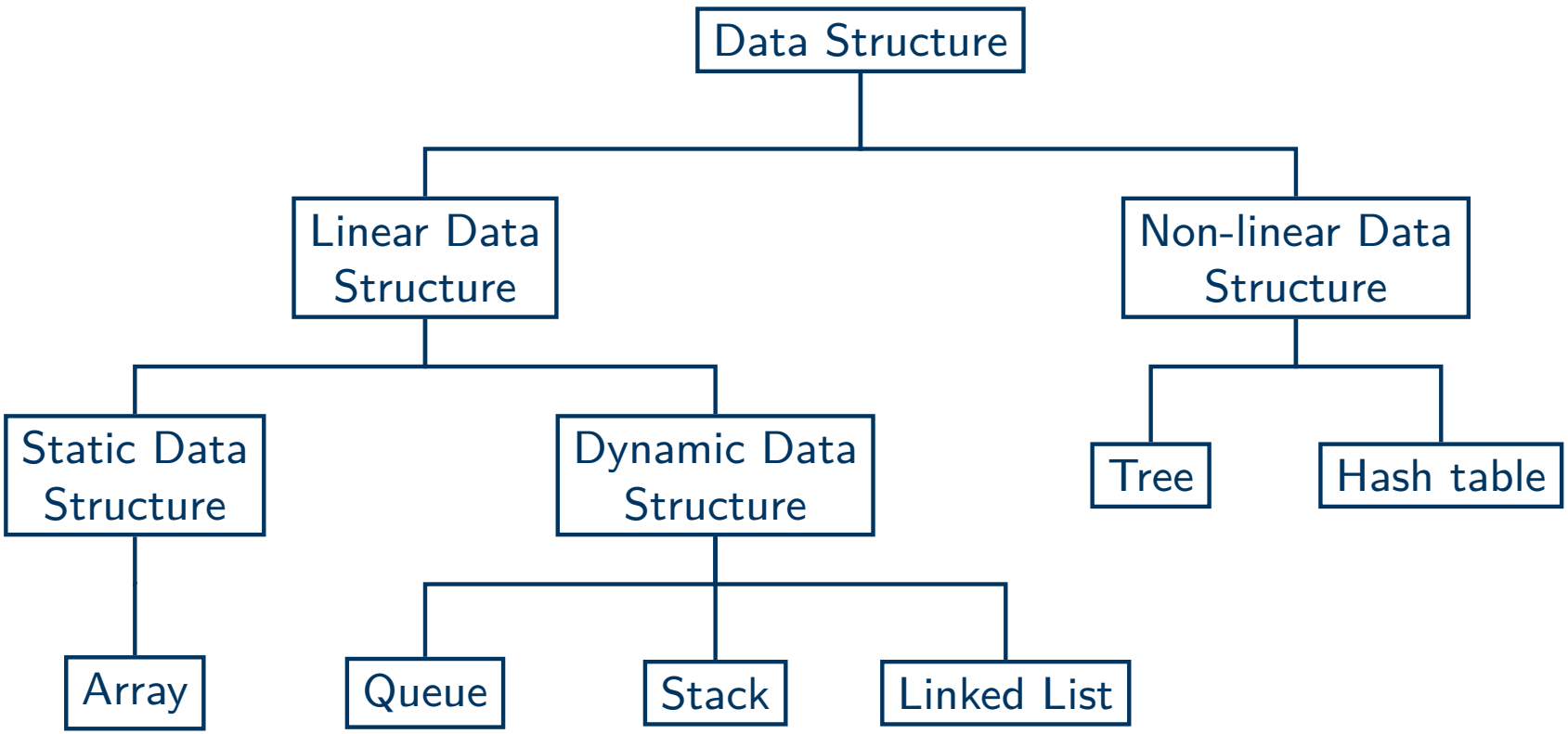
Data structures

Data structures are used to store and organize data. They are essential for efficient data access and form a basis of many programs. There exists two major subcategories:

- Linear Data Structures: the data is arranged sequentially, where each member is attached to the next element. Therefore, all elements can be traversed in one run
- Non-linear Data Structures: elements are not ordered sequentially, impossible to access all members in one run

Non-linear structures often utilize the memory more efficiently and the time complexity for operations such as searching a specific element is lower compared to linear data structures.

Data structures

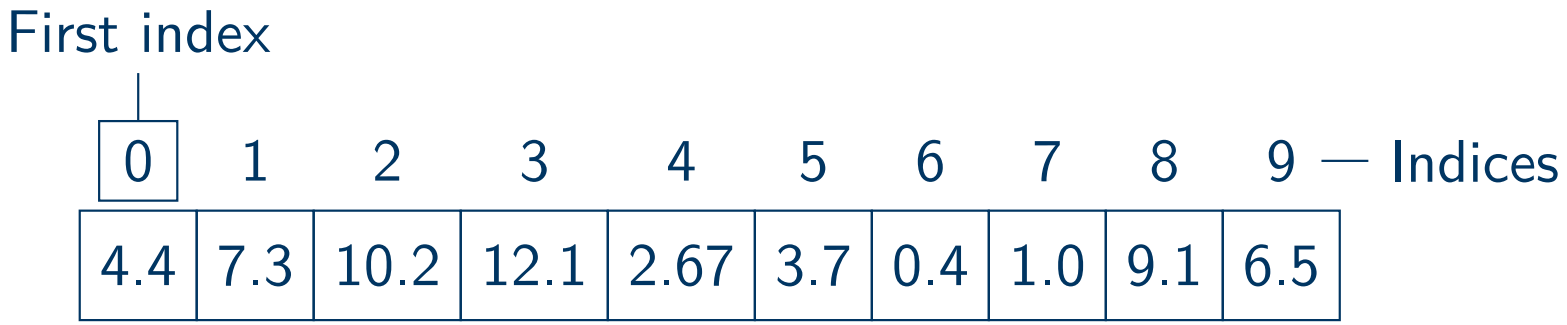


Linear data structures

Static data structure - array

An array is a collection of items of the same data type stored at contiguous memory locations:

- Elements are ordered and an array can be considered as a vector
- An array of size N is accessed via indices $0, \ldots, N-1$
- Initialized with a fixed size (static)
- Often used to implement other data structures



Static data structure - array

Time and space complexity

	Access	Search	Insertion	Deletion	Space
Average	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\mathcal{O}(N)$
Worst	$\mathcal{O}(1)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$

Usage

Since arrays can serve as vectors and matrices, they are important in many fields of engineering. Nonetheless, due to their static size and expensive insertion and deletion operations, there often exist better alternatives for software development.

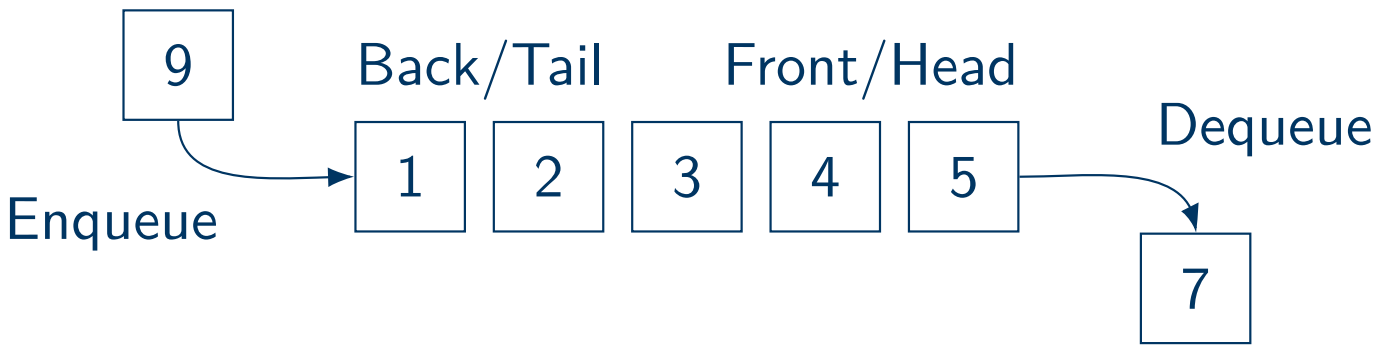
C++ STL implementation

The C++ Standard Template Library (STL) provides array/vector implementations with the vector template class and array template class (in C++11)

Dynamic data structure - queue

A queue is a sequence of entities which can be modified at both ends:

- It implements the first-in-first-out (FIFO) principle
- In a FIFO data structure, the first element added to the queue will be the first one to be removed
- Elements can only be added at the back (tail), called enqueue, and removed at the front (head), called dequeue
- Dynamic size throughout its lifetime
- Can contain different data types



Dynamic data structure - queue

Time and space complexity

	Access	Search	Insertion	Deletion	Space
Average	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(N)$
Worst	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$

Usage

Due to their flexible size and FIFO mechanism, queues are often used for sequential problems, for example in scheduling. There, the efficient insertion and deletion is utilized the most.

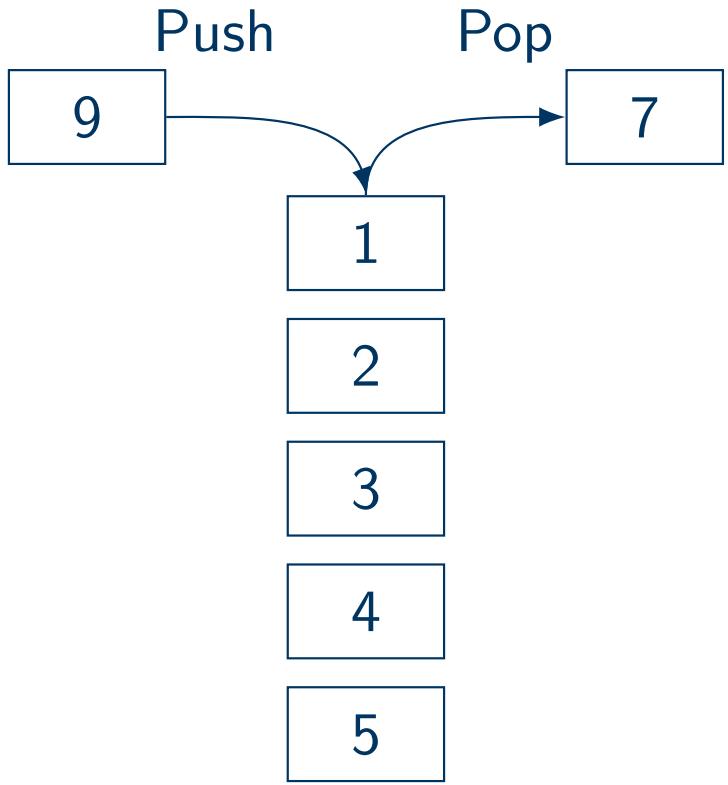
C++ STL implementation

The C++ Standard Template Library (STL) provides a queue template class.

Dynamic data structure - stack

A stack is a collection of items, where only one end, the top, can be accessed:

- It implements the last-in-first-out (LIFO) principle
- Therefore, the element inserted last will be removed first
- Two main operations, removing the topmost element, called pop, and adding a new element on top, called push
- Dynamic size throughout its lifetime
- Can contain different data types



Dynamic data structure - stack

Time and space complexity

	Access	Search	Insertion	Deletion	Space
Average	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(N)$
Worst	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$

Usage

Due to their flexible size and LIFO mechanism, stacks are often used for applications where the reverse order is needed, for example in backtracking or the static memory in C++.

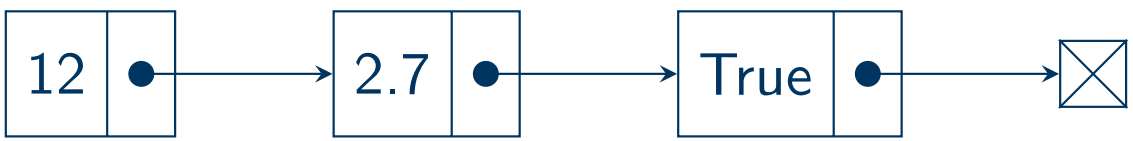
C++ STL implementation

The C++ Standard Template Library (STL) provides a stack template class.

Dynamic Data Structure - Linked List

A linked list consists of elements whose order is not given by their physical placement in memory:

- It is a collection of nodes, where each element points to the next
- Each node contains data and a link to the next node
- This allows for efficient insertion and removal of nodes at any position
- On the other hand, no indexing is possible and simple operations such as obtaining the last node of the list may require iterating through all list elements



Dynamic data structure - linked list

Time and space complexity

	Access	Search	Insertion	Deletion	Space
Average	$\Theta(N)$	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(N)$
Worst	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$

Usage

Linked lists are a good choice over arrays when their efficient insert and deletion mechanism is needed as well as the dynamic size.

C++ STL implementation

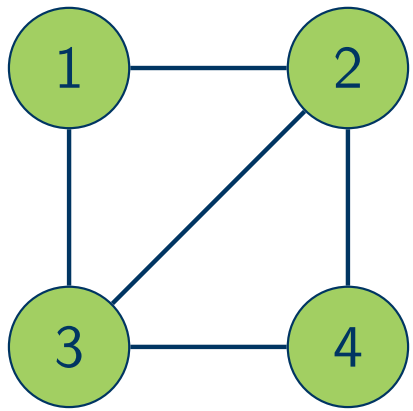
The C++ Standard Template Library (STL) provides a list template class.

Non-linear data structures

Graph

A graph $G = (V, E)$ is a non-linear data structure consisting of nodes V and edges E:

- The set of nodes V can be labelled or unlabelled
- The set of edges E describes the connections between nodes
- Graphs are the basis for many important structures and algorithms (e.g. trees)
- Example: $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$



There exists a wide variety of graphs.

Trees

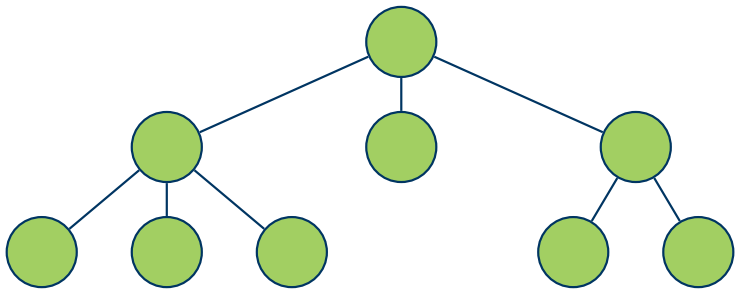
A tree data structure is a hierarchical graph, which focusses on efficient navigation and searching:

- Tree structures generally have a non-linear time complexity of $\mathcal{O}(\log(N))$ for their operations
- They consist of nodes and edges, where a node contains an item and pointers to its child nodes
- The topmost node is called root and each node can have arbitrary many children but only one parent
- A tree usually consists of elements of the same data type

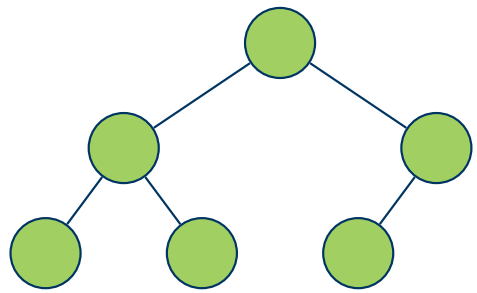
Tree examples

There exists a wide variety of tree structures:

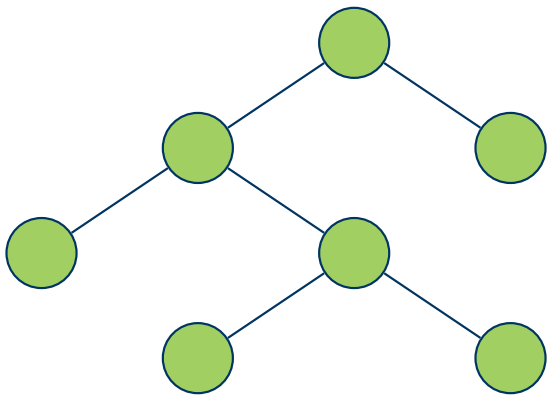
Generic tree with arbitrary many children



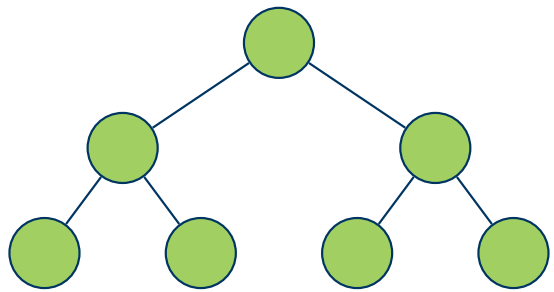
Generic binary tree, at most 2 children on each node



Full binary tree, every node has either 0 or 2 children



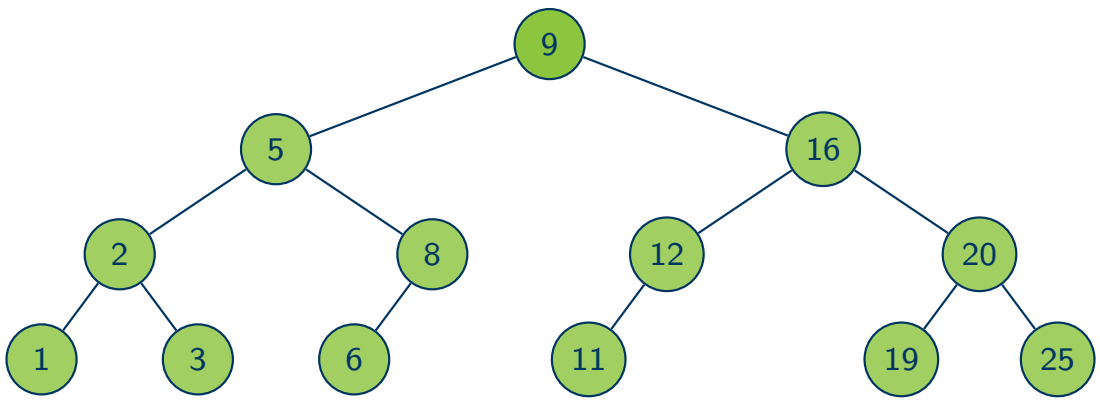
Perfect binary tree, completely filled



Binary search tree

A binary search tree (BST) is a binary tree data structure in which each node has at most two children:

- The value of each node is greater than all nodes of its left subtree and smaller than all nodes of its right subtree
- Insertion and deletion change the shape of the BST dynamically to ensure its ordering property
- The structure of the tree can be utilized to grand fast searching operations
- Dynamic size throughout its lifetime



Binary search tree

Time and space complexity

	Access	Search	Insertion	Deletion	Space
Average	$\Theta(\log(N))$	$\Theta(\log(N))$	$\Theta(\log(N))$	$\Theta(\log(N))$	$\mathcal{O}(N)$
Worst	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$

Usage

Underlying data structure used to implement many other non-linear structures, such as maps and sets. Also present in efficient sorting algorithms, for example quick sort and tree sort.

C++ STL implementation

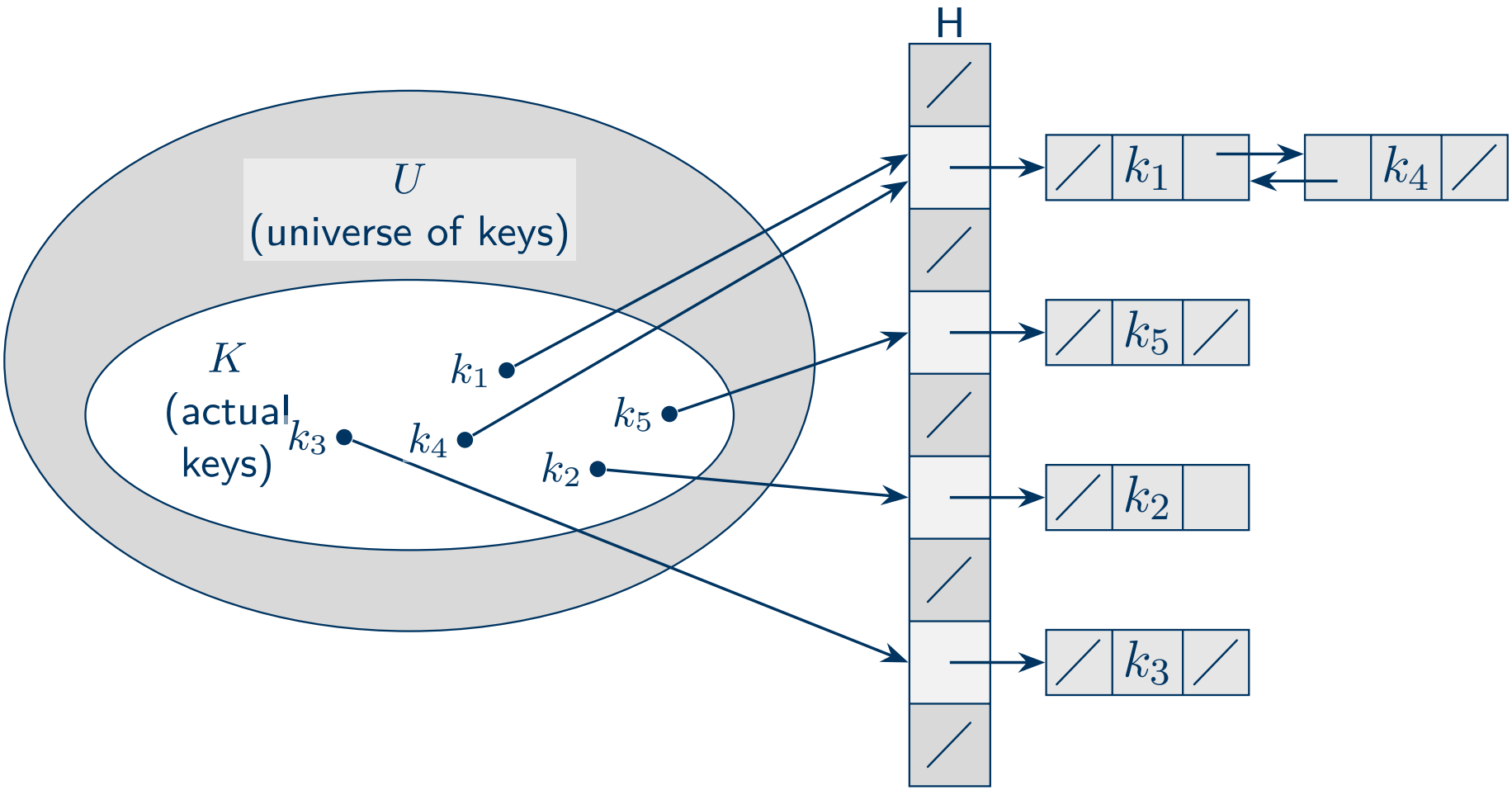
The C++ Standard Template Library (STL) does not provide a template class for BSTs.

Hash table

A hash table implements a map, that associates keys with values:

- A hash function is used to compute an index that refers to the respective memory cell
- This function couples keys to values
- A collision occurs when the hash function generates the same index for different keys and there exist several possibilities to resolve a collision
- A good function enables fast data access
- A hash table is not ordered and no random access is possible

Hash table



Hash table

Time and space complexity

	Access	Search	Insertion	Deletion	Space
Average	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\mathcal{O}(N)$
Worst	N/A	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$

Usage

Hash tables are extensively used in practice due to their, on average, efficient time complexity. Examples are databases indexing and password storage.

C++ STL implementation

The C++ Standard Template Library (STL) provides several implementations for hash tables, namely unorderedset and unorderedmap

Summary

	Array	Queue	Stack	Linked List	BST	Hash Table
Access	$\Theta(1)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(\log(N))$	N/A
Search	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$	$\Theta(\log(N))$	$\Theta(1)$
Insert	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log(N))$	$\Theta(1)$
Delete	$\Theta(N)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\log(N))$	$\Theta(1)$
Space	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Strength	Random access	FIFO	LIFO	Easy insert	Efficient search	Key-value map
Weakness	Static	No random access	No random access	No random access	Operations not always logarith-mic	Hash collisions
C++ STL	Vector/ Array	Queue	Stack	List	N/A	Un-ordered set

Git

Git

git is . . .

- a **V**ersion-**C**ontrol **S**ystem (**VCS**)
  - tracks changes in computer files over time
  - helps to coordinate work on a common project among multiple people
  - allows to access prior versions of a project, revert changes, review files
- a **distributed** version-control system
  - complete history mirrored on every local computer
  - allows to work offline, provides multiple backups
- **free** and **open-source** software
- developed since 2005 to maintain the Linux kernel
- the **widely used** version-control system in software development
  (alternatives: BitKeeper, Monotone, Mercurial, Bazaar,
  svn (subversion), cvs (Concurrent Versions System))

The name *git* – meaning *unpleasant person* in British English slang – was given by Linus Torvalds (creator of the Linux kernel) who is quoted as: "*I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'.*"

Sources, Literature, Hosting

Several **resources** and **literature** about git exist:

- **Official website**: git-scm.com
- Git **documentation**: git-scm.com/doc
- Git **book**: git-scm.com/book
  (*Pro Git* by S. Chacon, B. Straub)
- Git **cheat sheets**
  (summarizing reference cards for git commands)
- Git command **help**: `git --help`

Several **source code hosting facilities** for git projects:

- GitHub: github.com
- Bitbucket: bitbucket.org
- GitLab: gitlab.com
- ...

Terminology

Git is used to keep track of the changes to files of a project.

- **Project:** All files in a folder (incl. subfolders)
- **Working directory:** The project folder

Using Git, the full history of the development can be recorded partitioned into user-defined snapshots.

- **Commit:** Snapshot of the state of the project at a given point in time annotated with a descriptive message (Revision)
- **Staging:** Telling Git what files to include in the next commit
- **Repository:** History of all changes of a project (incl. metadata)

Each snapshot is identified by a SHA-1 checksum (hash) of it's content.

- **Hashvalue**: used as identifier for a commit

In general, it is only possible to add information to a repository – not to remove it. Git only stores modified information from commit to commit.

Features

git is a **distributed version-control system** with the following features:

- **Efficiency**
  Fast, scalable, and performance tested for large projects
- **No central server, distributed development**
  Every user owns a local copy of the whole version history
- **Cryptographic authentication of history**
  Commits stored using Hash-ID depending on the complete development history; older versions cannot be changed without notice
- **Non-linear development**
  Creation of new development branches and merging of several branches supported
- **Data transfer between repositories**
  Several transfer protocols allow for data transfer between repositories: `git://`, `ssh://`, `git+ssh://`, `http://`, ...
- **Toolkits**
  Several graphical tools provided, e.g. `gitk`, `git gui`

Basic Concepts and Commands

Configuration

Git can be configured using the git configure command:

```
git config <scope> user.name <name>
git config <scope> user.email <email-address>
```

[param]	<scope>	Scope control: --system, --global, --local
[param]	<name>	User name
[param]	<email-address>	Email address

- **name** and **email-address** are referenced in commits
- scope is system-wide (--system), every repository of a user (--global), or a single repository (--local)

```
$ git config --global user.name  UserName
$ git config --global user.email username@rub.de
$ git config --list
user.email=username@rub.de
user.name=UserName
:
:
```

Initializing a Repository

```
git init <directory>
```

[param] `<directory>` Name of project folder

- Converts <directory> into a repository
  (if folder does not exist, it is created first)
- If the argument <directory> is omitted, the current folder will be converted into a repository
- Basically, an invisible folder `.git` is created, where all snapshots and metadata is stored

```
$ git init myRepository
Initialized empty Git repository in C:/Users/UserName/myRepository/.git
```

Cloning an existing Repository

```
git clone <path/url> <directory>
```

[param] <path/url> Path or URL to existing repository

[param] <directory> Name for local repository (optional)

- Creates a full copy of the specified repository

```
$ git clone git://github.com/schacon/grit Copy
cloning into Copy...
remote: Enumeration objects: 4051, done.
remote:total 4051 (delta 0), reused 0 (delta 0), pack-reused 4051
Receiving objects: 100% (4051/4051), 2.04 MiB|1.89 MiB/s, done.
Resolving deltas: 100% (1465/1465), done.
```

Status of Repository

```
git status
```

- Shows status of working directory (branch, commit, ...)
- Shows status of files (untracked, modified, staged)
- Use option `-s` for a compressed view

```
git diff --staged
```

`[param]` `--staged` Show staged changes (optional)

- By default, shows all unstaged changes
- Using `--staged`, shows all staged changes
- Provides a detailed look into implemented changes

File states

The repository only keeps track of all files it knows about. There are thus two different kind of files:

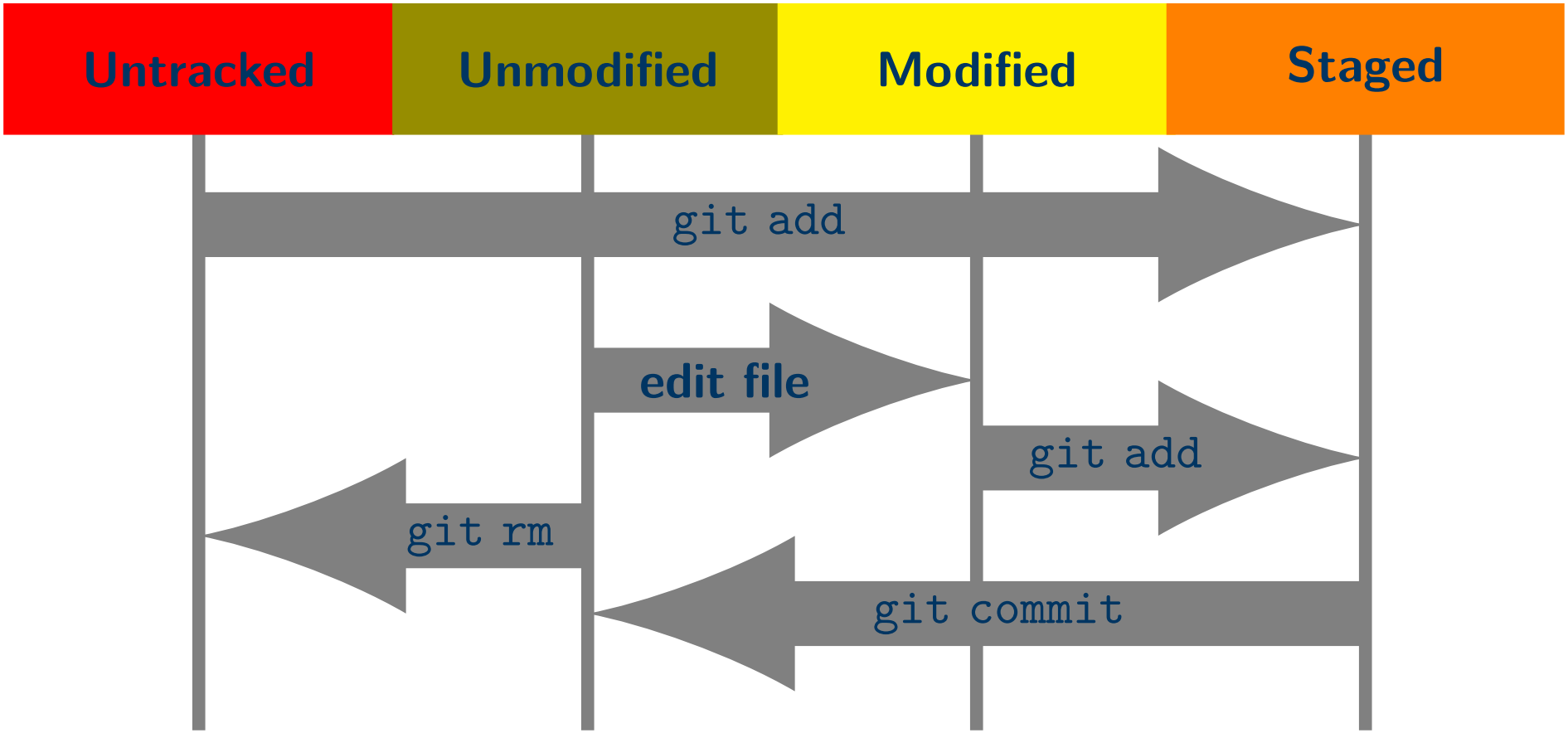
- **untracked:** files not under version control
- **tracked:** files under version control

A tracked file can have the following states:

- **unmodified:** not touched since last commit
- **modified:** changed but not staged for the next commit
- **staged:** marked in the current modified version for the next commit

File states

The state of a file is modified as shown below:



Staging Files

```
git add <file>/<directory>
```

[param] `<file>/<directory>` File or directory to stage

- Adds the **current** state of the file to the next commit
- File is in **staged** status afterwards
- An untracked file will be tracked afterwards

```
$ nano NewFile
$ git add NewFile
$ git status
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
    (use "git reset HEAD <file>." to unstage)
        new File: New File
```

Deleting and Moving Files

Deleting and moving of files has to be changed and committed.

```
git rm --cached <file>
```

[param] `<file>` File to remove in next commit

[param] `--cached` Only untrack file (optional)

```
git mv <src-file> <dest-file>
```

[param] `<src-file>` Source file

[param] `<dest-file>` Destination file

- `git rm` removes the file from working directory
- The `--cached` option will untrack, but not delete the file
- `git rm` stages the removal for the next commit
- `git mv` stages the file for commit after name change

Deleting and Moving Files

```
$ nano RemoveFile
$ git add RemoveFile
$ git rm --cached RemoveFile
rm "'rm RemoveFile'"
$ git status
On branch master
Your branch is up to date with "'origin/master'".
Changes to be committed:
    (use "git reset HEAD <file>..."to unstage)
        new file: NewFile
untracked files:
    (use "git add<file>..."to include in what will be committed)
        RemoveFile
```

- `git rm` will instantly stop tracking a staged file and will stage a committed file for deletion.

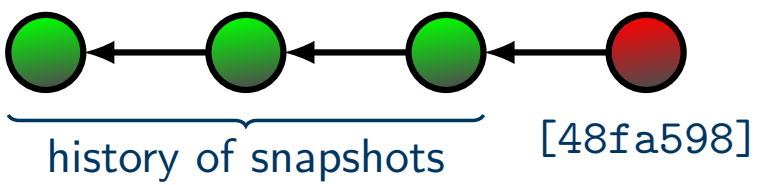
Taking Snapshots (commit)

```
git commit -m "<descriptive message>"
```

[param] `-m <"descriptive message">` Commit message (optional)

- Creates a snapshot (commit) of the project taking into account all currently staged files
- A descriptive message is stored with the commit
- The message can be specified on command line using `-m`. Otherwise, an editor opens to provide the message there

```
$ git commit -m "Test commit"
[master 48fa598] Test commit
1 file changed, 1 insertion(+)
create mode 100644 NewFile
```



Combined Staging and Committing

```
git commit -a -m "<descriptive message>"
```

[param]	-m <"descriptive message">	Commit message (optional)
[param]	-a	Add all modified

- Stages **all modified** files and commits afterwards

We can also create a file that indicates that certain class of files has to be ignored. This is implemented by creating a **.gitignore** file where the ignore pattern format conventions can be found. This file can be created by commands like cat and touch.

Stash uncommited Changes

Sometimes one wants to store unfinished work, without committing. To this end, `git stash` can save non-committed work in a stack.

```
git stash
```

- Stack up the uncommitted changes
- Leaves a clean working directory

```
git stash list
```

[param] `list` List stored stashes

```
$ git stash
Saved working directory and index state WIP on nB:3c92a80 fixed cli
    show bug
$ git stash list
stash@{0}: WIP on nB: 3c92a80 fixed cli show bug
```

Stash uncommited Changes

```
git stash apply <stash-name> --index
```

[param]	`apply`	Apply newest stash
[param]	`<stash-name>`	Optional name of stash to apply
[param]	`--index`	Option to restage changes

- `git stash apply` applies the newest stored stash (by default)
- Other stashs to apply can be chosen by <stash-name>

```
git stash drop <stash-name>
```

[param]	`drop <stash-name>`	Removes from stack

- Deletes the named stash

Stash uncommited Changes

```
git stash pop
```

[param] pop Apply newest stash

- pop applies and removes the stash from the stack
- It can be used as combination of apply followed by drop

```
$ git stash apply stash@{0} --index
<stdin>:8:new blank line at EON.
+
warning:1 line adds with "newRepo/master"
Changes to be committed:
    (use "git reset HEAD <file>..."to unstage)
    new file: NewFile
$ git stash drop stash@{0}
Dropped stash@{0} (9a1a26df2849bb796b457)
```

Showing the Commit History

```
git log
```

- Will show the last commits implemented on the project

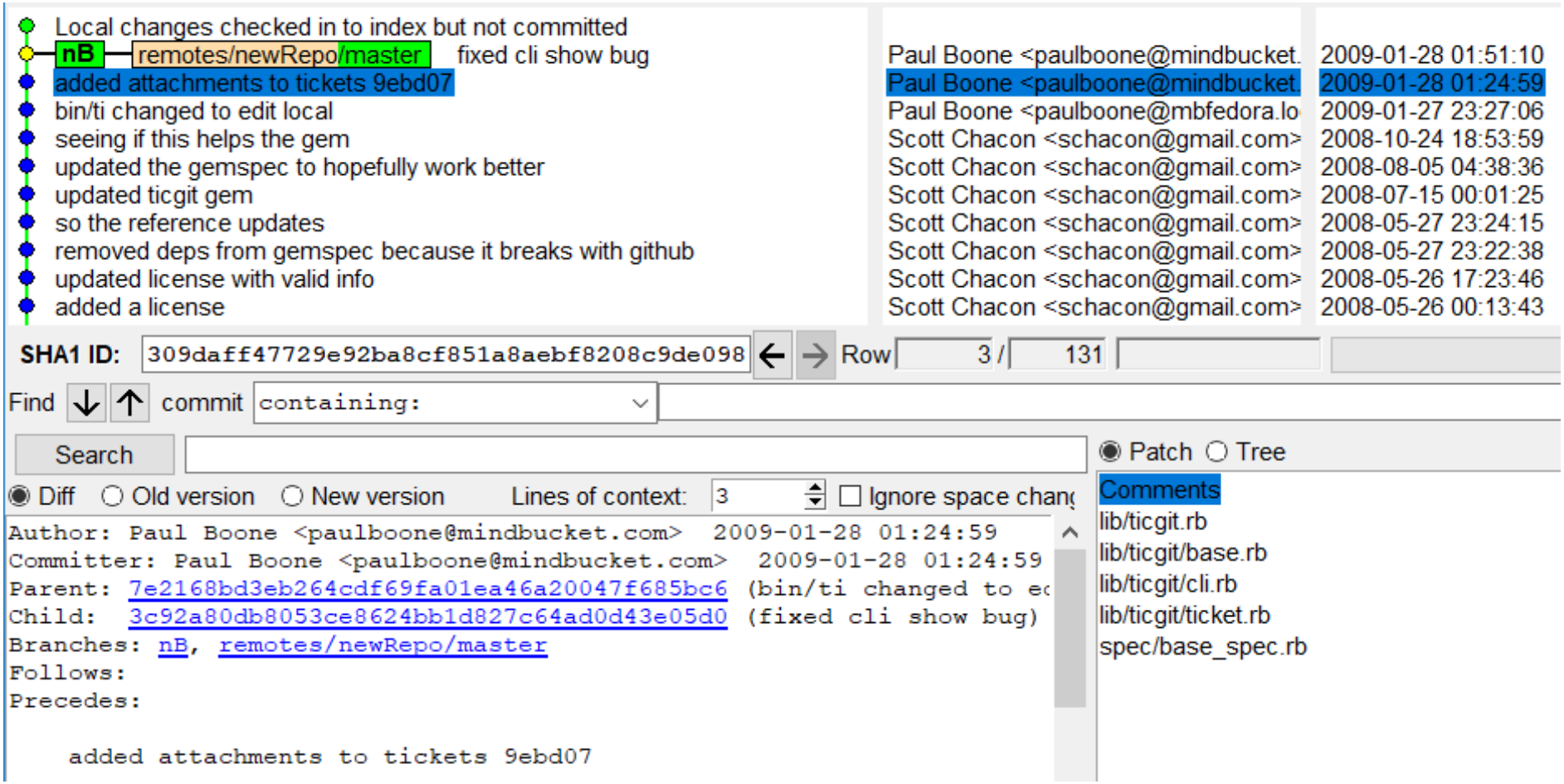
```
gitk
```

- Provides a user-friendly GUI for commit review

```
$ git log
commit 48fa59871e92f151dfec4127eafbd415550c077814 (HEAD ->master)
Author: UserName<username@.de>
Date:Tue Oct 30 09:24:46 2018 +0100
    Test commit
:
```

Showing the Commit History

The use of `gitk` provides the following GUI, showing the commit history, the comments, and the branching.



Undoing Changes

Undoing Changes

At any stage, changes to the project can be undone. For Git, most undoings will not destroy any information but rather produce a new snapshot the reverts unwanted changes. However, some commands may destroy some information and have to be used with care.

Depending on which state has to be undone, there are several options:

- **Amend the last commit by adding something**
- **Unmodify a modified file**
- **Unstage a staged file**
- **Revert committed changes**

Changes can be undone both on commits and on individual files.

Undoing Changes for Files

- To **unstage** a staged file, but not discard changes in working directory:

```
git reset HEAD <file>
```

[param] <file> File to unstage

- To **unmodify** a modified (but not staged) file, i.e. to discard changes in the working directory:

```
git checkout -- <file>
```

[param] <file> File to unmodify

Undoing Changes for whole Commits

```
git reset <mode> <commit>
```

[param] <mode> Mode

[param] <commit> Commit resetting to

- Resets to a specific <commit>, specified by: a SHA-1 Checksum, HEAD (most recent commit), HEAD~N (N previous commits), or by default (which is HEAD)
- The option tells what to reset (default is --mixed):
  - --soft: not modify staging area nor working area
  - --mixed: reset staging area, not modify working area
  - --hard: reset staging area and modify working area

```
$ git reset --hard HEAD~
HEAD is now at 0bd0c5f implement select_existing_bjects
```

Undoing Changes for whole Commits

```
git revert <commit>
```

[param] <commit> Commit to revert

- Reverts the changes that the <commit> introduced by adding a new commit to the history that applies the revers changes
- Requires the working directory to be clean (no modifications from the HEAD commit)

```
git clean -f
```

- Removes all untracked files from working directory

Branching

Branching

In Git, a **branch** is an independent line of development. This concept is pretty useful to allow for the development of new features in a separate (maybe experimental) setting and then **merging** the branch into the main development.

- A branch is a pointer to a commit
- As a standard, Git provides the user with a **master** branch
- Other branches can be created by the user

As a developer, one can only edit one branch at a time. The branch one is currently working on is indicated by the so-called `HEAD` **pointer**. By changing the `HEAD` pointer one can switch between branches and also take a look at previous commits in history.

Creation of Branches

```
git branch <Name>
```

[param] <Name> Branch to be created

- Creates a branch pointing to the same commit as the current branch
- The current branch is indicated by the HEAD pointer

```
git checkout <branch>
```

[param] <branch> Branch name

- Switches to branch <branch> (i.e. changes HEAD pointer)
- **Changes** the content of **working directory** to current branch state

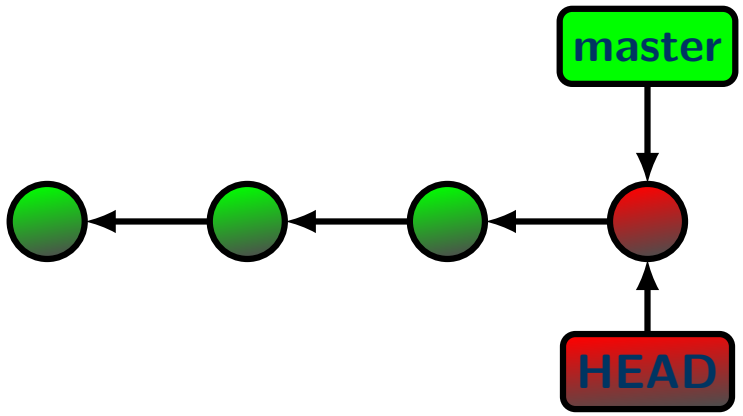
```
git checkout -b <branch>
```

[param] <branch> Branch name

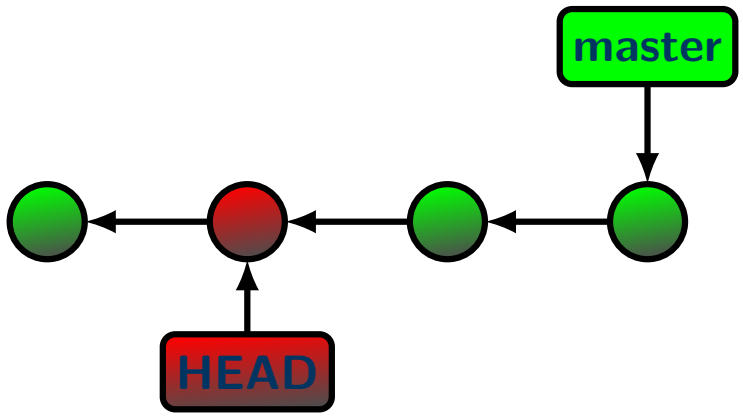
- Creates new branch <branch> and switches to that branch

Checkout of previous Commits

Usually, HEAD pointer and master branch coincide on the same commit.
(HEAD pointer will be indicated by red circle for a commit snapshot)



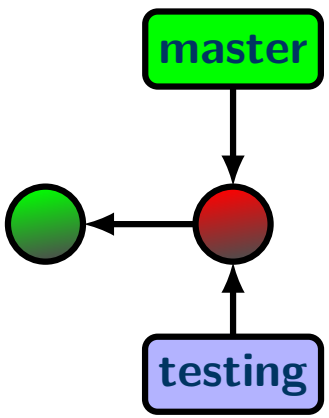
The checkout command can move the HEAD pointer to a different commit. In the working directory, files will correspond to the checked out commit. In this case, one is in a *no branch* status (`detached HEAD`).



Creation of Branches

The asterisk * will indicate to which branch HEAD is pointing to.

```
$ git branch testing
$ git branch
*master
 testing
$ git checkout testing
Switched to branch 'testing'
$ git branch
 master
*testing
```

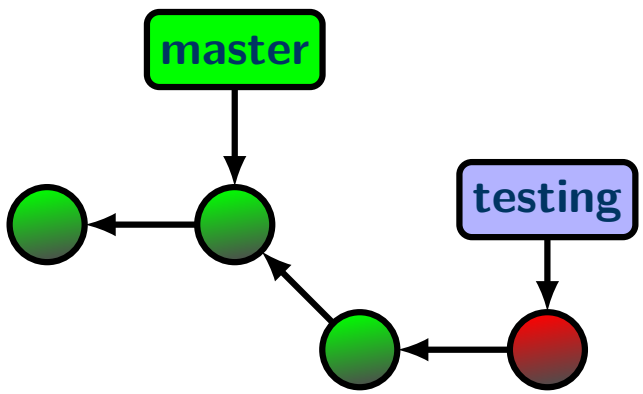


Creation of Branches

The commits that follow will be add to the history on the **testing** branch not on **master**.

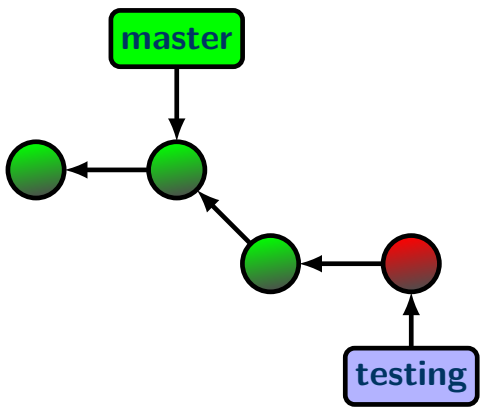
```
$ nano FileOnTesting
$ git add FileOnTesting
$ git commit -m '"Commit on testing"'
```

The created file will only be present while we are in this branch. Once we `checkout` to another branch, it will disappear from our working directory.

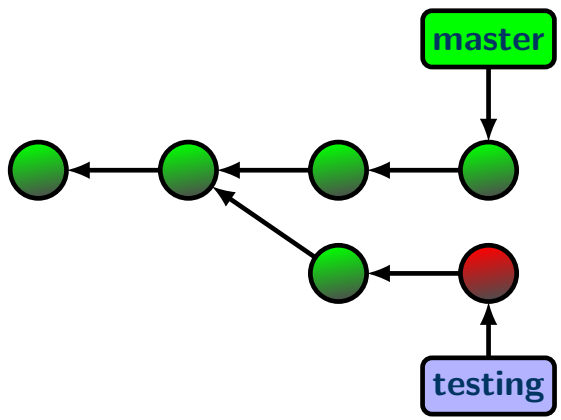


Merging

- The changes made in one branch can be incorporate into another branch. This is called **merging**
- Git implements different ways of merging:
  - **fast-forward**: both branches follow same history
  - **3-way**: history of both branches has diverged
- Avoid errors by merging branches with no un-committed changes



fast-forward merge



3-way merge

Merging Branches

```
git merge <branch>
```

[param] <branch> Branch to merge

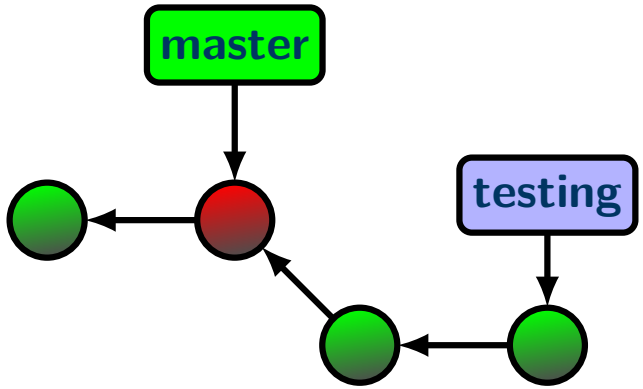
- Merges into the current branch

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
$ git merge testing
Updating e1e38bc..1385c4a
Fast-forward
 OnTesting | 1 +
 1 file changed,1 insertion(+)
 create mode 100644 OnTesting
```

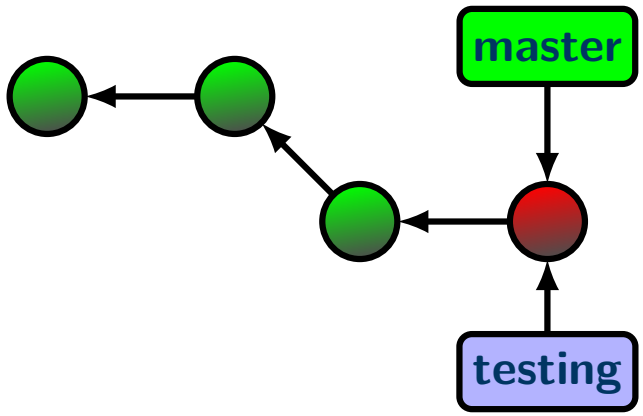
It can be verified that the file created on the new branch is now present in the master branch.

Fast-forward Merging

The feature branch `testing` forks from the unmodified `master` branch commit history, allowing for a *fast-forward* merge.

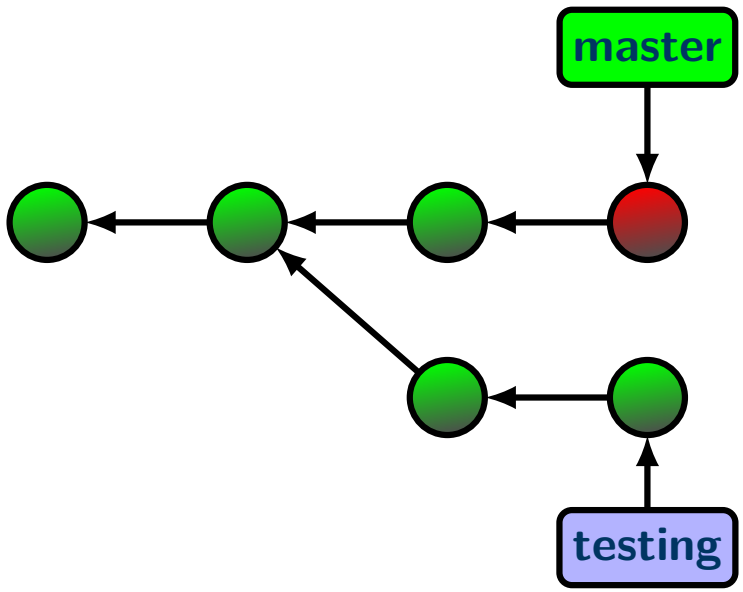


The merging of the feature branch `testing` into the `master` branch can be carried out linearly following the commit history.



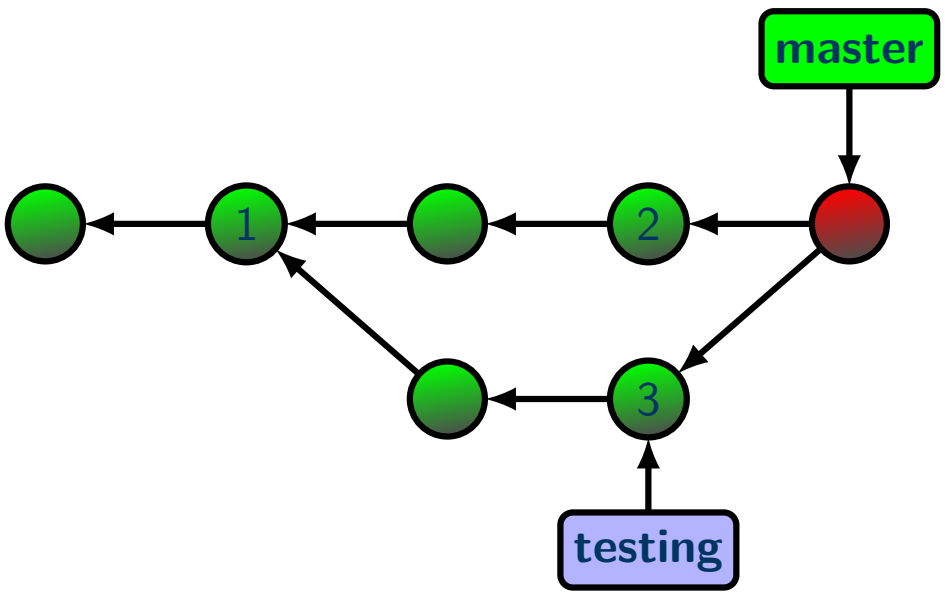
3-Way Merging

When the commit histories of 2 branches have diverged, the merging algorithm depends on a common anchestor commit. The merging can no longer be implemented using the fast-forward algorithm.



3-Way Merging

The merging will add a new commit, which depends on both branches, therefore the new merged commit has access to both commit histories. The algorithm will rely on 3 commits: the 2 parent commits and the common ancestor in the commit history. Because of this we call this method a *3-way merging*.



Managing Branches

```
git branch -v --mode
```

[param] `-v` Verbose flag

[param] `--mode` Mode option

- `git branch` used for viewing, control and review of branches
- `--mode`: choose `--merged` or `--no-merged`

```
git branch -d <branch>
```

[param] `-d <branch>` Branch to delete

- Deletes the branch -d <branch>

```
$ git branch -d testing
Deleted branch testing (was 13854c4a)
```

Merging Conflicts

Merge conflicts are caused by merging branches with different changes on the same file. Git will automatically stop the merging process and indicate the conflict. The merging then can be accomplished by:

- Use `git status` to see what is unmerged
- Solve accordingly for every file, use `git add` to mark as resolved
- Use `git commit` to finalize the attempted merge

```
$ git merge newBranch
Auto-merging newFile
CONFLICT (content):Merge conflict in newFile
Automatic merge failed:fix conflicts and then commit the result
$ git status
You have unmerged paths.
    (fix conflicts and run git commit)
    (use git merge ---abort to abort the merge)
Unmerged paths:
    (use git add <file> to mark resolution)
        both modified: newFile
```

Merging Conflicts

```
$ nano newFile
<<<<<<<HEAD
This is a new File
=======
This is not a new File
>>>>>>>newBranch
```

We have to implement the desired changes, e.g. we can choose the version we need from the both by deleting the other content.

```
This is a new File
```

After saving the content we finalize the merge with **git commit**, notice the use of **-a** for simplicity. We can modify the default commit message or accept it by simply closing the text editor.

```
$ git commit -a
[master 67fea5a] Merge branch newBranch
```

Remote Repositories

Adding Remote Repositories

A **remote repository** is a version of the project (a clone) at a **different location**. The location can be somewhere in the internet, some LAN server, or even a distinct location on your file system.

By cloning (`git clone`), automatically a remote repository is added (called **origin**). However, more remotes can be added manually:

```
git remote add <short-name> <path/url>
```

[param]	<short-name>	Short name
[param]	<path/url>	Origin URL or path of repository

- The <short-name> is used as reference

Inspecting Remote Repositories

```
git remote -v
```

[param] `-v` Show URL or Path (optional)

- without `-v` option: show **available remote** repositories
- with `-v` option: show corresponding URL in addition

```
git remote show <remote>
```

[param] `<remote>` Name of repository

- Shows detailed information about the remote repository <remote>
- E.g., shows branches used for `push` or `pull`

Adding Remote Repositories

After adding a remote repository, we have use `git show` to see which are the available branches.

```
$ git remote add newRepo git://github.com/paulboone/ticgit
$ git remote
newRepo
origin
$ git remote show newRepo
    Fetch URL: git://github.com/paulboone/ticgit
    Push  URL: git://github.com/paulboone/ticgit
    HEAD branch:master
    Remote branches:
        master tracked
        ticgit    tracked
    Local ref configured for ''git push'':
        master pushes to master (local out of date)
```

Copying from Remotes

```
git fetch <remote>
```

[param] <remote> Name of remote repository

- Downloads the changes from the repository (without merging)
- To incorporate the new changes, merging has to be done manually

```
git pull <remote>
```

[param] <remote> Name of remote repository

- Performs a `fetch` and followed by a `merge` into the local repository
- If the local repository has been **cloned**, then passing no <remote> argument will update from the origin of cloned remote

Tracking Remote Branches

```
git checkout -b <local-branch> <remote/branch-of-remote>
```

[param]	<local-branch>	Local branch to create
[param]	<remote/branch-of-remote>	Tracked remote branch

- Create a new branch in the local repository tracking the selected branch from the remote
- Being local, this branch can be used in the working directory

Managing Remote Branches

We use `git fetch` and `git checkout` to include the remote repository's files into one of our project's branches. It is important to understand that even if we fetch a remote repository, this does not provide us with usable copies of the branches. Instead, we have to create local branches to be able to work with the files.

```
$ git fetch newRepo
remote: Enumerating objects:634, done.
remote:Total 634 (delta 0), reused 0 (delta 0), pack-reused 634
Receiving objects: 100 \% (634/634), 88.92 KiB| 650.00 KiB/s, done.
Resolving deltas:100\% (261/261), done.
From git://github.com/paulboone/ticgit
* [new branch]  master -> newRepo/master
* [new branch]  ticgit    -> newRepo/ticgit
$ git checkout -b nB newRepo/master
```

It can be verified how the working directory changes it content as we `checkout` from one branch to the other.

Pushing changes to Remotes

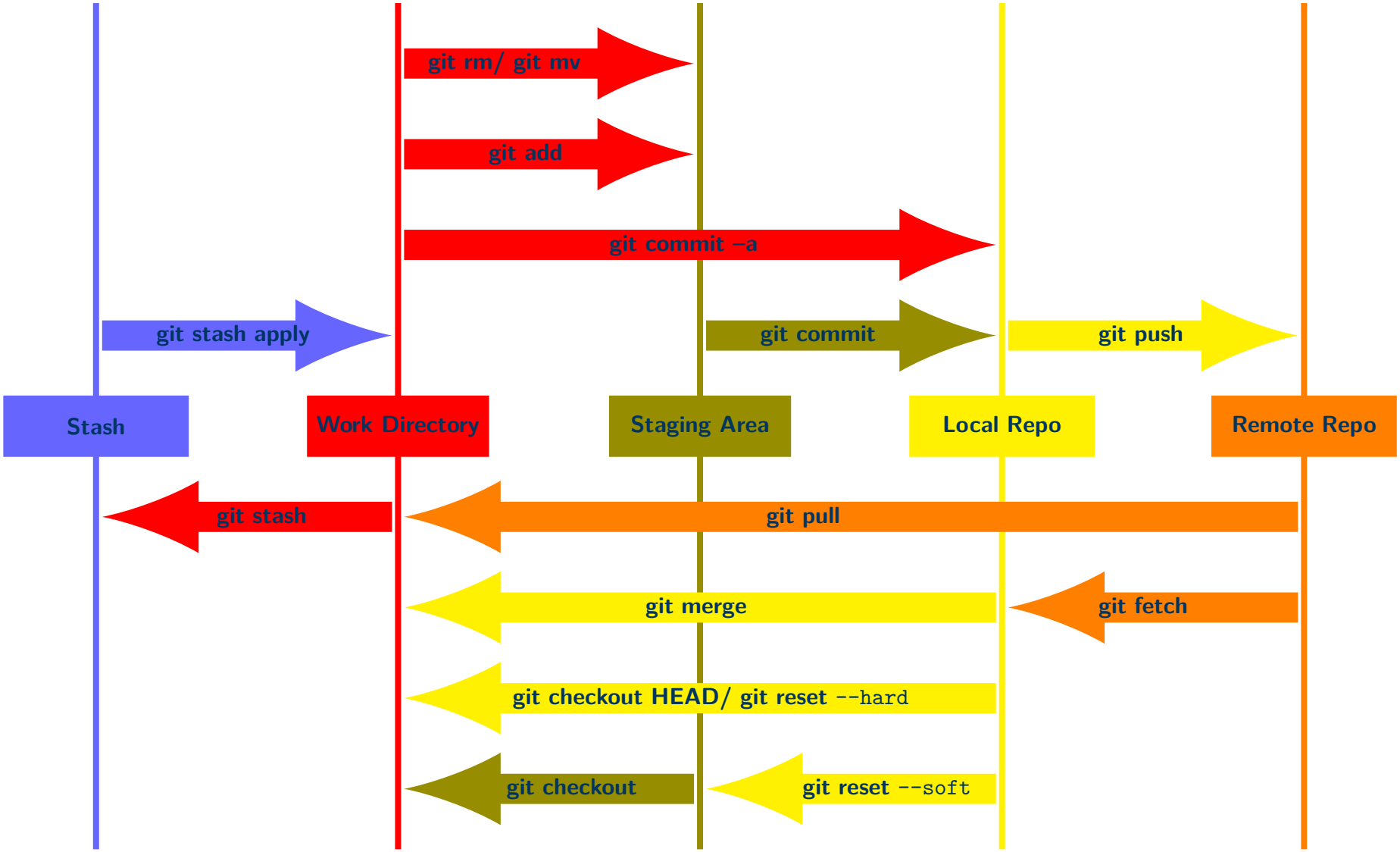
```
git push <remote> <branch>
```

[param] `<remote>` Name of remote repository

[param] `<branch>` Name of branch being pushed

- Contrary to `git fetch`, `git push` will write the branch to the remote
- The use of `git push` is subject to having the proper write access to the server

Overview



Containerization

Containerization

Motivation:

- Development and deployment of software is complicated because of
  - OS compatibility
  - Dependencies
  - Conflicts
  - Etc.
- Traditionally, the environment and dependencies must be manually configured

Containerization:

- Deploys software and all its dependencies in a bundle called container

Advantages:

- Portability
- Scalability
- Agility
- Etc.

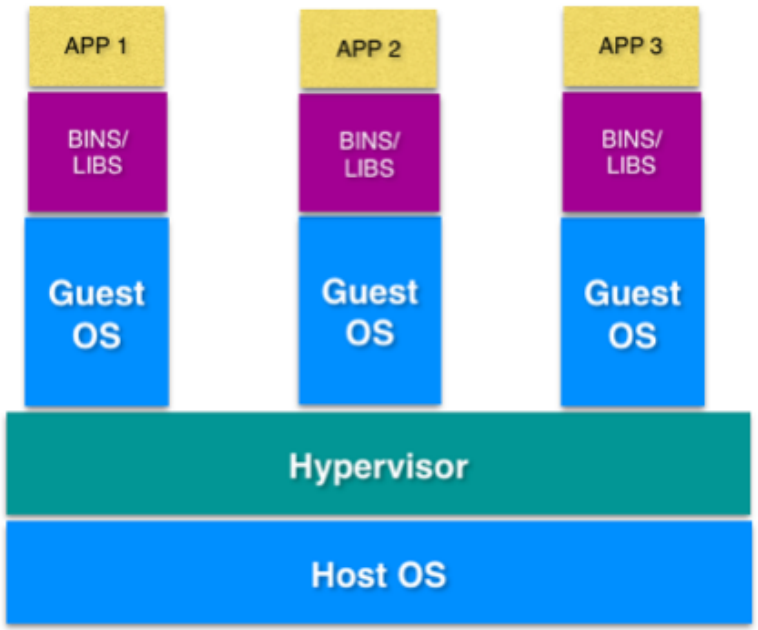
Docker

What is docker?

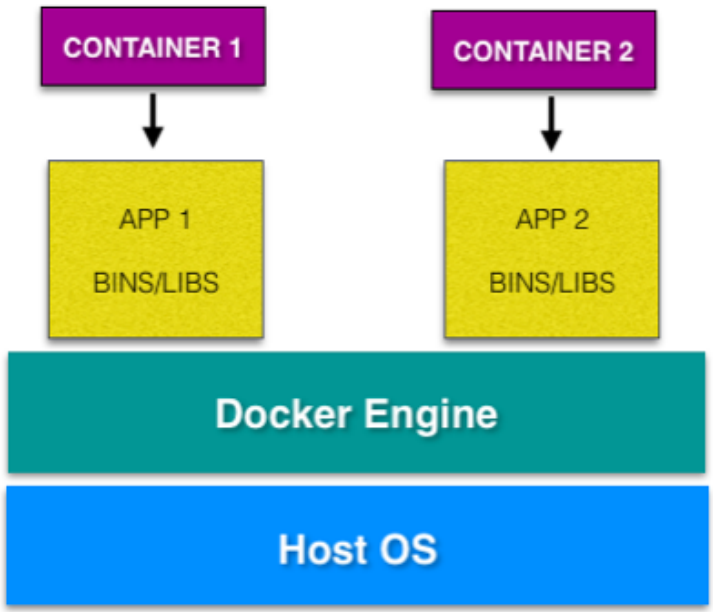
- Docker is an open-source project for development and deployment of software
- Docker uses OS-level virtualization to deliver software in self-contained `containers`
- `Containers` are standardized Docker units that include all the OS, dependencies and code required to run the target application
- Docker Engine hosts the containers and runs on Linux, MacOS and Windows

Docker vs. Virtual Machine

VM is based on a hypervisor layer, whereas Docker is based on a client/server layer through Docker Engine.



VIRTUAL MACHINE ARCHITECTURE



DOCKER ARCHITECTURE

Docker Advantages:

- Memory usage
- Performance

Docker Components

- Docker Engine (client and server): Command line tool based on REST API
- Docker image: Template with the instructions for the Docker container
- Docker container: Executable bundle of applications and dependencies
- Docker registry: Hosting location for Docker containers
- Docker compose: Allows running multiple containers as a single service
- Docker swarm: A services for managing multiple Docker nodes

Dockerfile

Dockerfile is a text file that includes the instructions to build a Docker image.
The `docker build` command builds the image from the Dockerfile
Sample Dockerfile:

```
# Creates a layer from the node:18-alpine Docker image
FROM node:18-alpine
# Copies files from the Docker client's current directory
COPY . /app
# Runs the command inside the image
RUN yarn install --production
# Specifies which commands to run in the container at start-up
CMD ["node", "src/index.js"]
# Exposes the port 3000 in the container
EXPOSE 3000
```

Useful Docker Commands

- `docker login`: Log in Docker
- `docker pull [image]`: Pull an image from the Docker registry
- `docker images -a`: List all local Docker images
- `docker image rm [image]`: Delete an image
- `docker run [options] [image] [commands]`: Run a container from a Docker image
- `docker container ls`: List running containers
- `docker rm [container]`: Delete a container
- `docker stop [container]`: Stop a running container
- `docker restart [container]`: Restart a running container
- `docker kill [container]`: Kill a running container
- `docker attach [container]`: Attach local command line to a running container
- `docker exec [options] [container] [command]`: Run a command inside a running container

Shell

Shell

A shell . . .

- is a user interface to the operating system
- is usually a command line interpreter
- provides tools to access other computer remotely
- is the best practice to work on supercomputers
- popular for Unix systems
- variants are available in Linux, MacOS and Windows

- Popular shell variant:
  - **bash** (**B**ourne **a**gain **sh**ell),
  - **zsh** (**Z sh**ell),
  - **tcsh** (**T**enex **C sh**ell)
  - **ksh** (**K**orn **sh**ell)
  - **fish** (**F**riendly **I**nteractive **sh**ell)

Navigating using a shell

Command	Effect
pwd	**p**rint **w**orking **d**irectory
cd	**c**hange **d**irectory to home folder
cd *some/folder*	change to relative directory *some/folder*
cd */some/folder*	change to absolute directory */some/folder*
cd ..	change to parent folder
cd -	change to last visited folder
ls	**ls**t files in folder
ls -a	list **a**ll (also hidden) files
ls -l	list files with **l**ong informations

- Folder separation in Linux is via / (opposed to \ on Windows)
- The home folder is given by $\sim$. Absolut path start with /
- Use 'Tab' for autocompletion, arrow keys to browse history

Creating files and folders

Command	Effect
mkdir *dirname*	**m**a**k**e **dir**ectory *dirname*
rmdir *dirname*	**r**e**m**ove **dir**ectory *dirname*
touch *filename*	create (empty) file *filename*
rm *filename*	**r**e**m**ove *filename*
rm -r *folder*	**r**e**m**ove **r**ecursively all files in *folder*
cp *srcFile to/path/destFile*	**c**o**p**y *srcFile to/path/destFile*
mv *srcFile to/path/destFile*	**m**o**v**e *srcFile to/path/destFile*

CMake

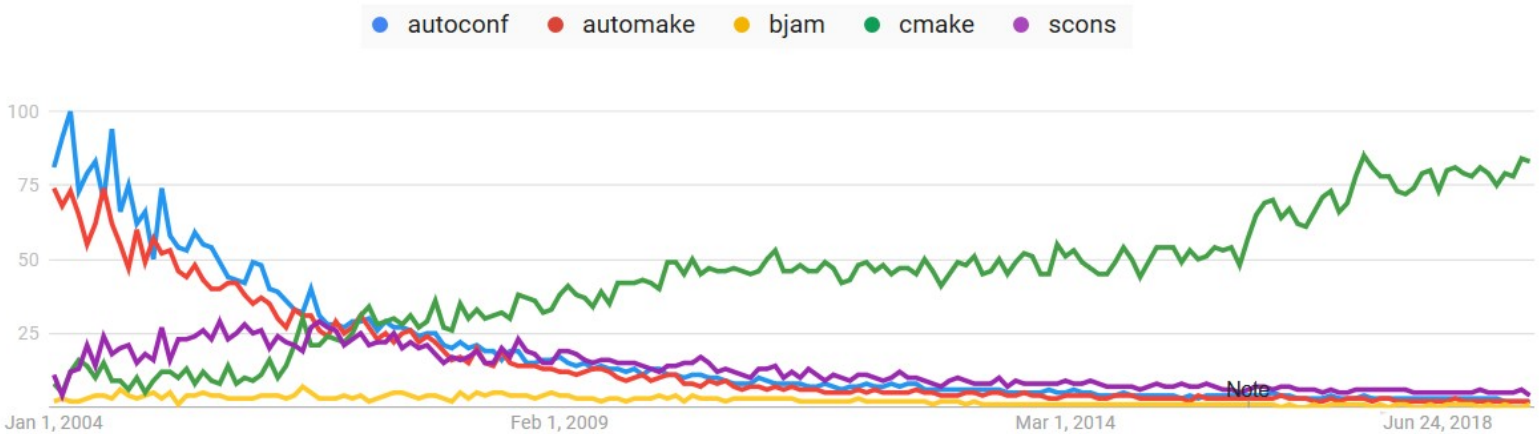
CMake

CMake is ...

- a **c**ross-platform **make**file generator
- **free** and **open-source**
- manages the software compilation process
- uses **configuration files** that are
  - **platform-independent** (runs on Linux, Windows, MacOS, ...)
  - **compiler-independent**
- **generates native makefiles** and **IDE projects**, e.g. for
  - Makefiles (GNU, Nmake, Borland, ...)
  - Eclipse
  - Visual Studio
  - XCode

Background

- Developed since 1999 by Kitware Inc. (initially for the ITK project)
- Matured and grown in popularity



Source: https://trends.google.com/trends/explore?date=all&q=autoconf,automake,bjam,cmake,scons

- One of the most used build systems today
  (e.g. used by Blender, Netflix, Second Life, MySQL, ...)
- Integrates nicely with **CPack** (packaging system), **CTest** (testing), and **CDash** (testing dashboard)

Literature

Several **resources** and **literature** about cmake exist:

- **Official website**: cmake.org
- CMake **documentation**:
  `https://cmake.org/cmake/help/latest/`
- CMake command **help**: `cmake --help`
- **Mastering CMake**
  by K. Martin, B. Hoffman (published by Kitware, Inc.)

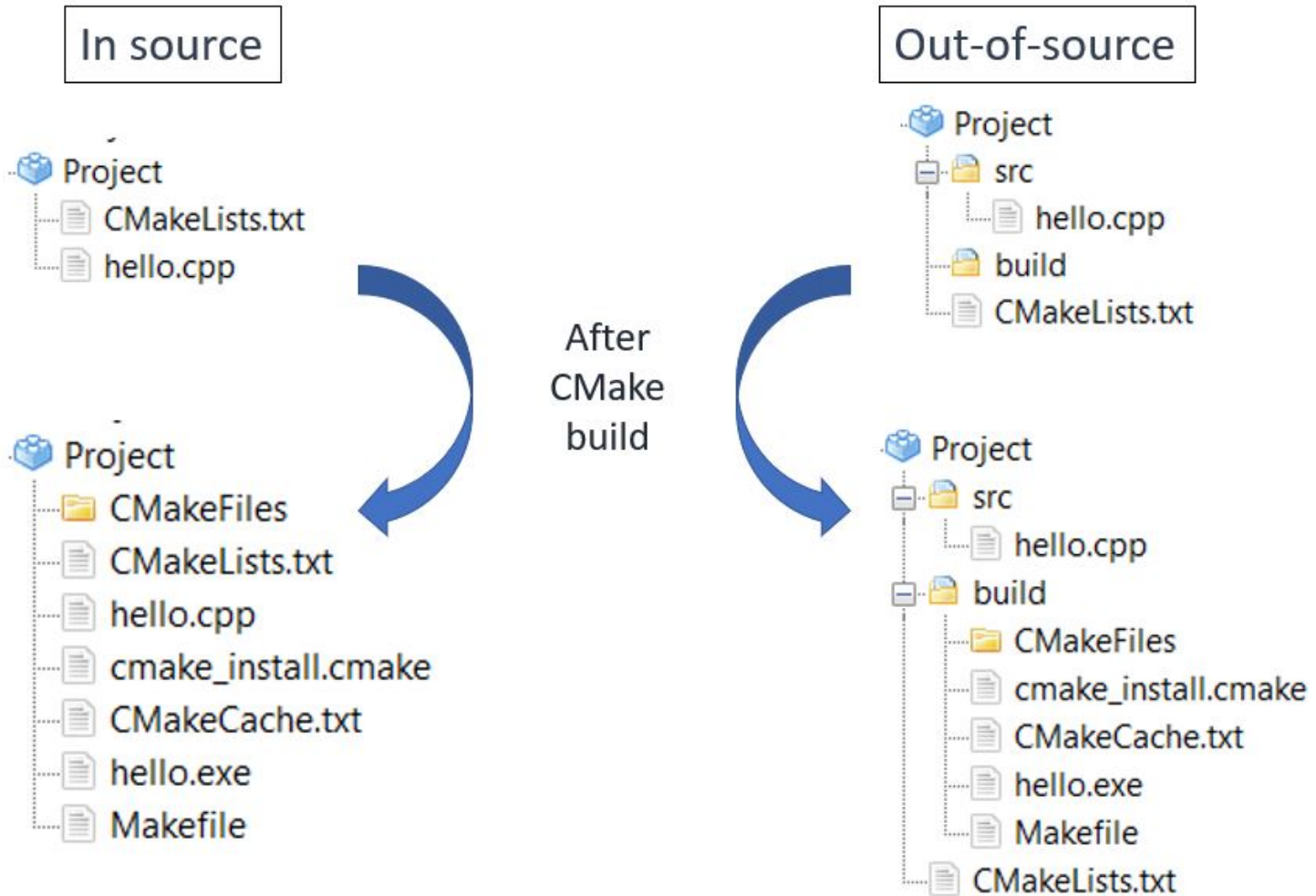
Features

- One language (**Cmake Language**) for all platforms
  - Windows, MacOS, Linux, UNIX variants, ...
- Supports large and involved projects
  - projects depending on **multiple libraries** / toolkits
  - complicated **directory hierarchies**
- Capable to **locate** required **headers** and **libraries**
- Allows **in-source** as well as **out-of-source** builds
  - target files built separately from source files
- Build process with CMake consists of **two stages**:
  - First, **create native build files** from CMake configuration files
  - Second, build **using** the platform's **native build tools**
- **Backward compatibility**
  - older scripts remain valid for newer versions
- **Version-control** system **friendly** development

Out-of-Source Builds

- Good practice to **keep source files** and **build files separated**
- Advantages:
  - Allows for **multiple build instances** at the same time
    - Build software for different conditions (debug, release, ...)
  - Allows for **cross-compilation** (build executable for different platform)
  - Easy and straightforward **cleanup** (removing separate build folder)
- CMake default: in-source
- Recommended: out-of-source
  Best practice: create source, build, and binary folders:
  - **src**: source codes (.cpp), header files (.h)
  - **build**: build files (.o)
  - **lib**: libraries (.a, .so, .dylib, ...)
  - **bin**: executables

Out-of-Source Builds



- Out-of-source places all build **artefacts** in **separate build** folder
- **Removing** the build directory, the **source** files remain **unaffected**

CMake Overview

CMake Language

- Language used in the **CMake source file**: `CMakeLists.txt`
- Only procedures, **no return values**
- **Control structures**
  - `if() ... elseif() ... else() ... endif()`
  - `foreach() ... endforeach()`
  - `while() ... endwhile()`
  - `break()`, `continue()`, `return()`
- **Functions**
  - Built-in (called *commands*)
  - User-defined (`function()`, `macro()`)
- **Modules**
  - CMake Language code stored in a `.cmake` file
  - Can be imported using `include()`

`CMakeLists.txt` file

A `CMakeLists.txt` file ...

- contains the set of instructions
- specifies the project's **build targets** (executables, libraries, ...)
- must be included in the top-level directory
- in addition, can be added to subdirectories for finer control (and be included in the build using `add_subdirectory()`)

- should at the very least contain these commands
  - `cmake_minimum_required(VERSION <x.x>)`
    - Sets the minimum required version for the project to `<x.x>`
  - `project(<name>)`
    - Sets the name of the project to `<name>`
    - The top-level `CMakeLists.txt` must contain this command
  - `add_executable(<exec> source1.cpp source2.cpp ...)`
    - Defines an executable target called `<exec>` from the source files

Running CMake on the commandline

```
cmake <path>
```

[in] `<path>` Path to the `CMakeLists.txt` file to execute

- Executes the CMake building process
  - runs cmake for the `CMakeLists.txt` specified by the relative path
  - detects required compilers, headers, ... (stored in a cache)
  - generates native build files
  - places `CMakeCache.txt`, Makefiles, and artefacts in current folder

```
$ cd project
$ touch ./CMakeLists.txt
$ cmake ./
-- The C compiler identification is gcc
-- The CXX compiler identification is g++
 [ ... ]
-- Configuring done
-- Generating done
-- Build files have been written to : /cygdrive/g/project
```

Example: HelloWorld using CMake

```cpp
#include <iostream>
int main(){
  std::cout << "Hello from C++ (built with CMake) !!" << std::endl;
}
```

hello.cpp

```
# Code in CMakelists.txt
cmake_minimum_required(VERSION 3.5)
project(HelloWorldProject)
message("Hello from CMake !!")
add_executable(hello src/hello.cpp)
```

CMakelists.txt

```
$ mkdir HelloWorld                  # create project directory
$ cd HelloWorld                     # move to directory
$ emacs CMakeLists.txt              # create CMakeLists.txt (as above)
$ mkdir src                         # create source folder
$ emacs src/hello.cpp               # create hello.cpp (as above)
```

Example: HelloWorld using CMake (building)

- Out-of-source build: Execute CMake in a `build` directory

```
$ mkdir build
$ cd build
$ cmake ..
 [... finding compilers ...]
Hello from CMake !!
-- Configuring done
-- Generating done
-- Build files have been written to: some/path/to/HelloWorld/build
```

- Run `make` to start the build process

```
$ ./make
Scanning dependencies of target hello
[ 50%] Building CXX object CMakeFiles/HelloWorld.dir/hello.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
```

Example: HelloWorld using CMake (result)

- The result is a project directory looking like this:

```
$ cd ..
$ tree -L 2
|-- CMakeLists.txt                    # CMakeLists
|
|-- src                               # source folder
|   |-- hello.cpp                     # source file
|
|-- build                             # build folder
    |-- CMakeCache.txt                # CMake cache
    |-- CMakeFiles                    # other cmake artefacts
    |-- Makefile                      # UNIX Makefile
    |-- cmake_install.cmake
    |-- hello                         # executable binary
```

- Run the built executable file `hello`

```
$ ./build/hello
Hello from C++ (built with CMake) !!
```

CMake Language and Commands

Displaying a message, Comments

```
message("message to display")
```

- Prints a message on the command line

Example:

```
message("Hello World !!")
```

```
$ cmake ./
Hello World !!
```

- Comments are specified as:

```
# This is a line comment

#[[ With brackets, it is a ...
      #  ... multiline comment ]]
```

Declaring variables

```
set(<variable> <value1> ..  <valueN>)
```

- Declares a (normal) variable
- Variable names are case sensitive
- All variables stored internally as strings
- Lists internally converted to one single string separated by ';'

Example:

```
set(abc "123") # Defines a string value
set(sourcefiles a.cpp b.cpp c.cpp) # Declares a list of files
set(sourcefiles "a.cpp;b.cpp;c.cpp") # The same as above
```

```
unset(<variable>)
```

- Deletes a variable

Example:

```
unset(abc)
```

Declaring a cache variable

```
set(<variable> "<value>" CACHE <type> "<string>" FORCE)
```

- [-] `<type>` STRING, BOOL, FILEPATH, PATH
- [-] `<string>` Message displayed in CMake GUI
- [-] `FORCE` Forces overwrite (optional)

- Assigns a value to a so-called **cache variable**
- **Cache variables** ...
  - are stored in the **CMake cache file**: `CMakeCache.txt`
  - **persist** across CMake runs
  - are **not overwritten** if already existing
    (i.e. only set in first run by default; use `FORCE` to overwrite)

Example:

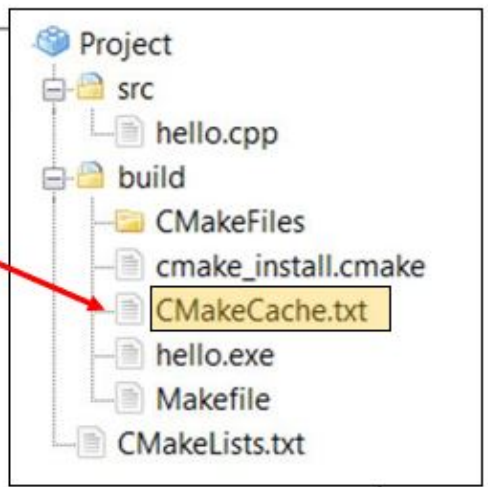
```
set(abc "123" CACHE STRING "Message" FORCE)
set(bol "ON" CACHE BOOL "Setting a boolean" FORCE)
set(dir "./dir" CACHE PATH "" FORCE)
set(filedir "C:/file.txt" CACHE FILEPATH "" FORCE)
```

CMakeCache.txt

- Cache build up in first CMake run (**configuring**)
- Cache can be **fine-tuned manually**
  - adjustment in `CMakeCache.txt` text file directly
  - using a CMake text based user interface (`ccmake` command)
  - using a CMake GUI (`cmake-gui` command)
  - using command line arguments (`cmake -D...`)
- Remove whole build to clean all cache variables

```
# Code in CMakeLists.txt
...
set(abc "123" CACHE STRING "This is an internal variable" FORCE)
...
```

```
# Code in CMakeCache.txt
...
//Path to a program.
CMAKE_AR:FILEPATH=/usr/bin/ar.exe

//This is an internal variable
abc:STRING=123

//Enable/Disable color output during build.
CMAKE_COLOR_MAKEFILE:BOOL=ON
    ...
```



Option variables

```
option(<variable> "<string>" [initial value])
```

- Shortcut to declare a cache variable with a boolean value
- Default initial value is `OFF`

Example:

```
option(DEUTSCH "" OFF)
if(DEUTSCH)
  message("Hallo Welt!")
else(DEUTSCH)
  message("Hello World!")
endif(DEUTSCH)
```

Output:

```
Hello World!
```

Referencing a variable, Scoping

```
${<variable>}
```

- Extracts the value of the variable

Example:

```
set(SomeVariable "abc")
message("Hello ${SomeVariable}")
```

```
Hello abc
```

A variable value is **evaluated** as follows (in order of precedence):

- Definition within current function
- Definition in current `CMakeLists.txt` (or parent)
- Definition in `CMakeCache.txt`
- Empty string if not found

if ... else

```
if(expression)
  # ...
elseif(expression2)
  # ...
else(expression)
  # ...
endif(expression)
```

- A **true** `expression` is given by the following values:
  `1`, `ON`, `YES`, `TRUE`, `Y`, or a non-zero number

```
set(abc "1")
if(abc)
  message("abc is a true variable")
else(abc)
  message("abc is a false variable")
endif(abc)
```

Expressions

CMake expression	Description
`NOT exp`	True if the expression is false
`<expr1> AND <expr2>`	True if both expressions are true
`<expr1> OR <expr2>`	True if either expression is true
`EXISTS directory`	True if the named file or directory exists
`var1 EQUAL var2`	True if var1 == var2
`var1 LESS var2`	True if var1 < var2
`var1 GREATER var2`	True if var1 > var2

For full list of expressions, refer here
`https://cmake.org/cmake/help/latest/command/if.html`

Built-in CMake commands

- **Commands** are **pre-defined functions** in the CMake language
- Reference for the **list of commands**:

  `https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html`

- Command names are **case-insensitive** (but arguments and variables are)

```
set(SomeVariable ON)
SET(SomeVariable ON)   # both equivalent
```

- Arguments are **space separated** and can spread multiple lines

```
set(var a.cpp b.cpp c.cpp)
set(var a.cpp
        b.cpp
        c.cpp)   # both equivalent
```

Specifying binary targets

```
add_executable(<exec> <source1> <source2> ...  )
```

- Defines the executable target <exec> by compiling the listed sources <source1>, <source2>, ...
- File ending automatically deduced:

  <exec>.exe (Windows) or <exec> (Linux, MacOS)

```
add_library(<lib> <type> <source1> <source2> ...  )
```

[-] <type> STATIC or SHARED

- Defines the library target <lib> from the list of source files
- Creates a static (STATIC, default) or shared (SHARED) library
- File ending automatically deduced:
  - static: <lib>.lib (Windows), <lib>.a (MacOS, Linux)
  - dynamic: <lib>.dll (Win), <lib>.dylib (MacOS), <lib>.so (Linux)

Specifying required libraries

```
target_link_libraries(<target> <lib1> ...  <libN>)
```

- [-] `<target>` Target file
- [-] `<lib>` Libraries

- Connects a target file to its dependent libraries
- Needs the target file and the libraries to be defined prior
  - Typically called after `add_executable()` and `add_library()`

Example: Executable target (project) depending on two libraries

```
add_library(lib1 source1.cpp source2.cpp)
add_library(lib2 source3.cpp source4.cpp)

add_executable(project main.cpp)

target_link_libraries(project lib1 lib2)
```

Handling includes

```
include_directories(dir1 dir2 ...)
```

- Adds directories where header files can be found
- Path of `dir` is relative to the location of the current `CMakeLists.txt`

```
find_package(<package>)
```

- Finds and loads settings from an external project
- If the package `<package>` is found,
  - the variable `<package>_FOUND` will have a value of `TRUE`
  - CMake will create the necessary environment variables containing information on the package's source files, libraries, and its dependencies

CMake Internal Variables

- **Built-in, pre-defined variables** in the CMake language
- Have **default values**, unless defined by user
- Reference for the **list of internal variables**:

  `https://cmake.org/cmake/help/latest/manual/cmake-variables.7.html`

- By convention: **all uppercase** and start with `CMAKE_...`
- Used for several **purposes**:
  - general informations (e.g. paths, ...)
  - information about the system
  - compile behavior (build type, shared libraries, ...)
  - compiler settings
  - ...
- Prominent **example**: `CMAKE_BUILD_TYPE`

  (with values: `Debug`, `Release`, `RelWithDebInfo`, `MinSizeRel`, ...)

Path variables

`CMAKE_RUNTIME_OUTPUT_DIRECTORY`

- Path where all the runtime target files (e.g., `.exe`) would be stored
- Typically a build/bin folder

`CMAKE_LIBRARY_OUTPUT_DIRECTORY`

- Path where all the runtime library target files would be stored

`CMAKE_ARCHIVE_OUTPUT_DIRECTORY`

- Similar to `CMAKE_LIBRARY_OUTPUT_DIRECTORY`, but in a static library

`CMAKE_CURRENT_LIST_FILE`

- Contains the path to the current CMake list file

C++ Compiler variable

`CMAKE_CXX_COMPILER`

- Specifies the desired C++ compiler by the user
- Note: The value of the variable will be stored in the cmake cache and persists for subsequent cmake runs
- List of inputs for C++ compilers:

C++ Compiler	Compiler IDs
C++	`c++`
G++	`g++`
GNU	`GNU`
Microsoft Visual Studio	`MSVC`
Clang	`Clang`

Full list of compiler IDs: `https://cmake.org/cmake/help/latest/manual/cmake-compile-features.7.html`

C++ compiler flags

`CMAKE_CXX_FLAGS`

- Stores list of flags to be used by the compiler

Example: Setting the compiler to be C++11 compatible

```
# Code in CMakeLists.txt
...
set(CMAKE_CXX_FLAGS "-std=c++11")
...
```

What is being implicitly executed on the commandline by CMake:

```
$ g++.exe -std=c++11 -o /.../hello.cpp.o -c /.../hello.cpp
```

Command line Invocations

Setting Cache Entries via command line

```
cmake <path> -D<var>=<value> ...
```

- [-] `<path>` path to CMakeLists.txt
- [-] `<var>` variable name
- [-] `<value>` value to assign

- Creates or updates a cmake cache entry

```
...
option(SomeVar ON)  # We set an initial value of ON
message("Variable is now ${SomeVar}")
...
```

```
$ cmake ./ -DSomeVar=OFF
Variable is now OFF
-- Configuring done
-- Generating done
-- Build files have been written to : /cygdrive/g/projectdir
```

Choosing a Compiler

```
cmake <path> -DCMAKE_CXX_COMPILER=compiler
```

- Choosing a compiler from the commandline
- Alters the compiler variable in the cache

```
$ cmake ./ -DCMAKE_CXX_COMPILER=g++
    ...
-- Check for working CXX compiler: /usr/bin/g++.exe
-- Check for working CXX compiler: /usr/bin/g++.exe -- works
    ...
```

```
CXX=compiler cmake <path>
```

- Alternative: set environment variable in bash for `CXX`
- CMake respects bash hint (but value stored in cache for first run only)

```
$ CXX=clang++ cmake ./
    ...
```

Example

- Case: To run the program in parallel mode with the g++ compiler
  - If in parallel, include the flag `-fopenmp` to run OpenMP
  - If in serial, include the flag `-O1` for optimization

```
# Code in CMakeLists.txt
...
option(PARALLEL OFF)

message("Parallel mode is ${PARALLEL}")
message("Compiler is ${CMAKE_CXX_COMPILER}")
if (PARALLEL)
    message("Running in parallel using OpenMP")
    set(CMAKE_CXX_FLAGS "-fopenmp")
else(PARALLEL)
    message("Running in serial with optimization")
    set(CMAKE_CXX_FLAGS "-O1")
endif(PARALLEL)
...
```

Example

```
$ cmake ./ -DPARALLEL=ON -DCMAKE_CXX_COMPILER=g++
Parallel mode is ON
Compiler is /usr/bin/g++.exe
Running in parallel using OpenMP
-- Configuring done
-- Generating done
-- Build files have been written to: /cygdrive/g/cygwin/Home/hello

$ make VERBOSE=1
...
/usr/bin/g++.exe -fopenmp -o /.../hello.cpp.o -c /.../hello.cpp
...
```

- For Makefiles, use `VERBOSE=1` to see full compiler call

Topics not covered

There are several aspects of CMake not covered in this tutorial:

- Installing (`install()`, ...)
- Testing (`add_test()`, ...)
- Packaging (create installable packages)
- Toolchains (especially for cross-compiling)
- Compile features (CMake 3.1+)
- Generators (create native build setup for IDE, ...), e.g.
  - `cmake -G"Unix Makefiles"`: standard **UNIX makefiles** (default)
  - `cmake -G"Ninja"`: **Ninja** (build system) files
  - `cmake -G"Eclipse CDT4 - Unix Makefiles"`: **Eclipse** project
  - `cmake -G"Visual Studio XX YYYY"`: **Visual Studio** project