C++ Compact Course

Till Niese (Z 705)
till.niese@uni.kn

Alexander Artiga Gonzalez (Z 712)
alexander.artiga-gonzalez@uni.kn

9 April 2018

Universität Konstanz

graphics.uni.kn

Organizational and Scope

Sessions

- Day 1
  - Overview (CMake, “Hello World!”)
  - Preprocessor, Compiler, Linker
  - STD Library
  - Input-/Output-Streams
- Day 2
  - Stack and Heap
  - Pointer
  - Classes
- Day 3
  - const, constexpr
- Day 4
  - Overloading
  - Templates, Inline
  - Default parameters
  - Lambda functions
- Day 5
  - assert, static
  - Compiler flags
  - ?

CMake (www.cmake.org)

CMake is an **open-source**, **cross-platform** family of tools designed to **build**, **test** and **package** software.
CMake is used to control the software compilation process using simple **platform and compiler independent configuration files**, and generate **native makefiles and workspaces** that can be used in the compiler environment of your choice.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.7)
project(HelloWorld)

set(CMAKE_CXX_STANDARD 14)

set(SOURCE_FILES main.cpp)
add_executable(HelloWorld ${SOURCE_FILES})
```

Invoke from command-line:

```
$cmake <Path to dir with CMakeLists.txt>
$make
```

Popular IDEs with CMake support (selection).

IDE: Integrated Development Environment

	CLion	QtCreator	Visual Studio	KDevelop	XCode
Linux	x	x		x	
Windows	x	x	x	(x)	
Mac OS X	x	x			x
Free	30 days	x	x	x	x

Note: Using CLion or QtCreator on Windows requires Cygwin (www.cygwin.com) or MinGW (www.mingw.org) installed.

CMake + Visual Studio (Windows)

1. **open CMake (gui)**
2. "Where is the source code" → select project folder
3. "Where to build binaries" → select build folder (e.g <Project folder>/build)
4. run "Configure"
5. select "Visual Studio XY 20XY" as "generator" and "Finish"
6. run "Generate"
7. **open "ProjectName.sln" from build folder with "Visual Studio"**
8. set "ProjectName" as StartUp Project (right click on it)
9. build release or debug, compile, run and enjoy

Similar for **Xcode**.
CLion, **KDevelop** and **QtCreator** offer direct import (select CMakeLists.txt directory).

Hello world!

```cpp
#include <iostream>

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

Hello world!

The compiler will only know identifiers that have been declared before usage.

```
#include <iostream>

void show_hello() {
    std::cout << "Hello world!" << std::endl;
}

int main(){
   show_hello();
   return 0;
}
```

Hello world!

Compiler will report: Use of undeclared identifier `show_hello`, if the an identifier was used before declaration or definition.

```
#include <iostream>

int main(){
   show_hello();
   return 0;
}

void show_hello() {
    std::cout << "Hello world!" << std::endl;
}
```

Hello world!

Especially on windows, the following will keep the console window open if your program is executed by Visual Studio or similar.

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    std::cout << "Press enter to continue...";
    std::cin.ignore(); // waits for input
    return 0;
}
```

(In Visual Studio, `Strg+F5` also keeps the console window open.)

Declaration and Definition

A **declaration** introduces an identifier and describes its type for the compiler. This allows the compiler to use it before it is defined.

```
double f(int, double);
class foo;
```

Declaration and Definition

A **definition** actually instantiates/implements this **identifier**. It's what the linker needs in order to link references to those entities. These are definitions corresponding to the above declarations:

```
double f(int i, double d) {return i+d;}
class foo {};
```

Hello world!

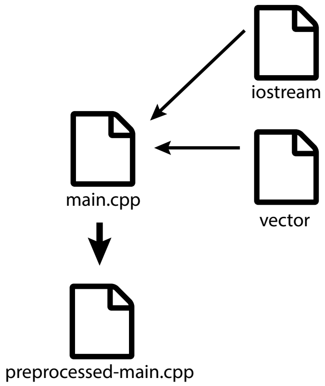
```
#include <iostream>

void show_hello();

int main(){
   show_hello();
   return 0;
}

void show_hello() {
    std::cout << "Hello world!" << std::endl;
}
```

Preprocessor, Compiler and Linker

Preprocessor

Searches for instructions like: `#include`, `#endif`, `#if`, `#define`, `#ifdef`, ...

- `#include` will *copy* the content of the file referenced by include.
- `#if`, `#ifdef`, ... are use to conditionally include/exclude code for the compiler for the current compilation unit.
- `#define` is used to define a preprocessor variable (can also be passed as compiler flag)

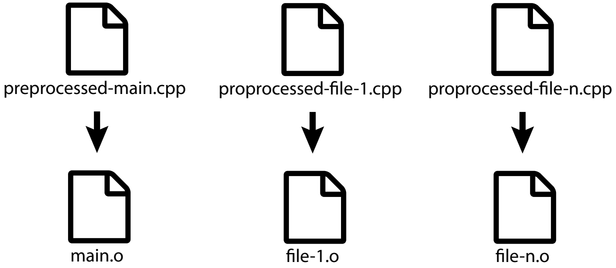


Preprocessor, Compiler and Linker

Compiler

The Compiler will compile the source code that was composed by the preprocessing step, into machine code instruction.

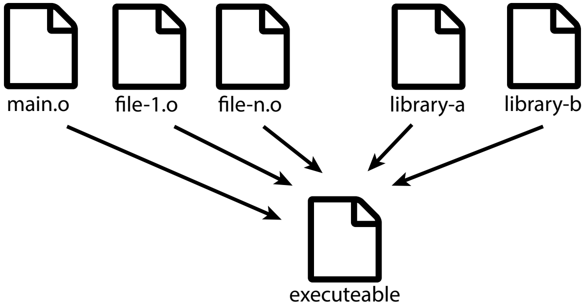
Each *.cpp* file will result in one **translation unit**



Preprocessor, Compiler and Linker

Linker

The linker will compose all generated **translation unit** into one executable. The linker will check if all **definitions** for each **declaration** exists, and ensures that there are no duplicate **declaration**.



Datatypes

Integral Datatypes

- `char`, `unsigned char` (at least 4Bit)
- `short`, `unsigned short` (at least 8Bit)
- `int`, `unsigned int` (at least 16bit)
- `long`, `unsigned long` (at least 32Bit)
- `long long` (at least 64Bit)

Floatingpoint Datatypes

- `float` (at least 32Bit), `double` (at least 64Bit)

Other Datatypes

- `bool`
- `auto` type will be automatically deduced from its initializer.

Datatypes

```
auto x1 = 1; // x1 will be an int
auto x2 = 1L; // x2 will be a long
auto x3 = 1.f; // x3 will be a float
auto x4 = 1.; // x4 will be a double
auto mc = MyClass();
```

The type of a variable defined with `auto` will not change during its lifetime:

```
auto mc = MyClass();
mc = 1.; // not valid, because mc is of type MyClass
```

`auto` is especially useful for **template** based or **lambda** functions. It would also allow to change the used datatype as long as they are compatible in the given scenario.
But it could harm readability because it is not always obvious what type will be held by the given variable.

Castings

C-Style-Casting:

```
int i = (int)f;
```

Disadvantage:

- C style cast are not checked at compile-time, and can fail at runtime.

Castings

C++-Style-Casting:

- `static_cast`
- `dynamic_cast`
- `reinterpret_cast`, `const_cast` do not use them until you are sure what they mean.

Disadvantage:

- It is more to write.

Advantages:

- It is easier to find typecast in the code.
- Intentions are conveyed much better

Implicit-Casting

```
int x = 2;
float y = 0.5;
int i = x*y;
```

is equivalent to

```
int x = 2;
float y = 0.5;
int i = static_cast<int>( static_cast<float>(x) * y );
```

Implicit-Casting

Could result in unexpected behaviour:

```
int x = 0;
float f = 0.8f;
x += f;
x += f;
```

`x` will be 0

STD Library

The std-library contains a huge number of classes and utility functions. The documentation can be found here http://en.cppreference.com/w/.

Often used headers:

- `iostream` input- and output-streams like `std::cout`
- `cmath` common math functions

Data types and containers that will be used often:

- `vector` is a variable length container with random access
- `array` is fixed length container with random access
- `list` is a variable length linked list
- `map` is a variable length container with key value pairs
- `string` is container for strings

Including headers of the std library is done without the `.h` suffix `#include <string>`

Container and Strings

In tutorials or books you might find the usage of strings or array like objects that looks like this:

```
int anIntArray[] = {0, 1, 2, 3, 4};
int *anotherIntArray = new int[10];
const char * aString = "some string";
```

If you write your code never use `new[]` but use those instead:

```
std::array<int, 5> anIntArray = {0, 1, 2, 3, 4};
std::array<int, 10> anotherIntArray;
std::string aString = "some string";
```

Containers and Strings

Advantages using std containers and strings over c-style containers:

- Memory management is handled by the container
- A large number of utility functions, for e.g. transforming, sorting, data, ...
- compatibility with modern 3rd party libraries

Containers

To use containers you have to define the data-type hold by the containers using **template arguments**.

```
std::array<int, 5> anIntArray;
std::vector<float> aFloatVector;
std::list<double> aDoubleList;
```

All containers have methods like `size` and `empty`. Containers with a dynamic size have methods like `push_back`, `pop_back`, `push_front` and `pop_front`.

The number of methods a std complex type provides by itself is limited to the important methods required by such a data type.

Iterator

All containers of the standard library offer access to the stored elements via iterators. Iterators are a uniform way to traverse a set of elements without having to know how they are stored.

- The `begin()` method returns an iterator that points to the first element of the set
- The `end()` method returns an iterator that points to the *past-the-last* element of the set

Depending on the type of container, there may also be iterators that allow the elements to be traversed backwards:

- The `rbegin()` method returns an iterator that points to the last element of the set
- The `rend()` method returns an iterator that points to the reverse *past-the-last* element of the set

Iterator

- An iterator can be manipulated using ++ this will move the iterator to the next element in the set.
- Iterators can be compared using the == operator
- If an iterator points to the last element in the set and ++ is called then the iterator will be equal to `end()`
- To get the element the iterator is pointing to you need to derefence the iterator using the * operator

Iterator

To iterate over a set of element a `for` loop can be used:

```
std::vector<int> vec = {10,20,30};
for( auto curr = vec.begin() ; it != vec.end() ; it++ ) {
  std::cout << (*it) << std::endl;
}
```

In a `for` loop you have exact control in which range you want to iterate. However, when you only want to iterate over all elements, you can also use the range base loop:

```
std::vector<int> vec = {10,20,30};
for( auto &elm : vec ) {
  std::cout << elm << std::endl;
}
```

Iterator

Especially with iterators you can see how useful the `auto` keyword is.
Without the auto keyword, the type of iterator must be specified completely:

```
std::vector<int> vec = {10,20,30};
for( std::vector<int>::iterator curr = vec.begin() ; it != vec.end() ; it++ ) {
  std::cout << (*it) << std::endl;
}
```

In case of a `std::vector` this is still relatively simple, in case of more complex libraries, this can quickly become very difficult to understand.

Iterator

The iteration over a `std::map` differs a bit, because the iterator does not point directly to the element, but to the key value pair. The pair has two members:

- `first` that represents the *key*
- `second` that represents the *value*

```
std::map<int, std::string> m =
  {
    { 1, "one" },
    { 2, "two" }
  };

for( auto &pair : m ) {
  std::cout << pair.first << " " << pair.second << std::endl;
}
```

Iterator

In addition to iterators, containers might offer additional options for accessing the elements. `std::vector` and `std::map` allow to access stored values using the `[]` `operator`:

```
std::vector<int> vec = {10,20,30};
std::cout << vec[0] << std::endl; // 10
vec[0] = 15;
std::cout << vec[0] << std::endl; // 15
```

Iterator

In case of a `std::map` you have to know, that if there is no entry for the key, a default element of this type is created first. Because of that you should not use the `[]` operator to insert elements into a map. You should use `insert()` or `emplace()` instead.

```
    std::map<int,int> m;
    m.insert(std::make_pair(1/*key*/,2/*value*/));
    m.insert(std::pair<int,int>(1,3)); //same as with make_pair
    m.emplace(1,4); //in-place
```

`insert()` and `emplace()` do nothing, if there is already an element with that key.
To change the value of an existing key, use `[]` operator, `at()` or `find()`.

```
    m[1] = 5; //inserts element, if key doesn't exist
    m.insert_or_assign(1,6); //inserts element, if key doesn't exist (C++17)
    m.at(1) = 7; //throws exception, if key doesn't exist

    auto result = m.find(1);
    if (result != m.end()) //does nothing, if key doesn't exist
        result->second = 7;
```

Helper-Functions

All additional functionality is provided by helper functions. Those helper functions can be used with most data type that conform to the interface of the data types in the std library.

The header `algorithm` (en.cppreference.com/w/cpp/algorithm) contains functions like:

- `find`
- `erase`
- `sort`

Helper-Functions

The `std::find` function can be used to check if an element is within a `std::vector`

```
int n = 2;
std::vector<int> v{0, 1, 2, 3, 4};

auto result = std::find(v.begin(), v.end(), n);

if (result != v.end()) {
    std::cout << "v contains: " << n << std::endl;
} else {
    std::cout << "v does not contain: " << n << std::endl;
}
```

`std::find` returns an **iterator** that points to the first found element that equals to `n`. Or `v.end()` if `n` was not found

Output- and Input-Streams

A streaming operator is used read or write data from or to streams:

- « writes data to a stream
- » reads data from a stream

Output-Streams

Writing to the stream `std::cout`:

```
#include <iostream>

void show_hello() {
    std::cout << "Hello world!" << std::endl;
}
```

Writing to a file:

```
#include <fstream>

void write_to_file() {
    ofstream myfile;
    myfile.open ("thefile.txt");
    myfile << "Hello world!" << std::endl;
}
```

Input-Streams

Reading from a file stream:

```
#include <fstream>

void read_from_file() {
  std::ifstream infile("thefile.txt");
  int a, b;

  if( infile.is_open() ) {
    infile >> a;
    infile >> b;
  } else {
    std::cout << "could not open file." << std::endl;
  }
}
```

Input-Streams - Error Handling

- Until `C++11`: If extraction fails (e.g. if a letter was entered where a digit is expected), value is left unmodified and failbit is set.
- Since `C++11`: If extraction fails, zero is written to value and failbit is set.

The `std::istream operator»` functions return their left argument by convention (in case of std::cin again std::cin), thus allowing the following:

```
int a, b, c;
infile >> a >> b; // (infile >> a) >> b:

while(std::cin >> c) {
  std::cout << c << std::endl;
}
```

The latter makes use of `operator bool` (since `C++11`, `operator void*()` until `C++11`), that returns true in case of no errors and false otherwise (e.g. if eofbit or failbit is set).

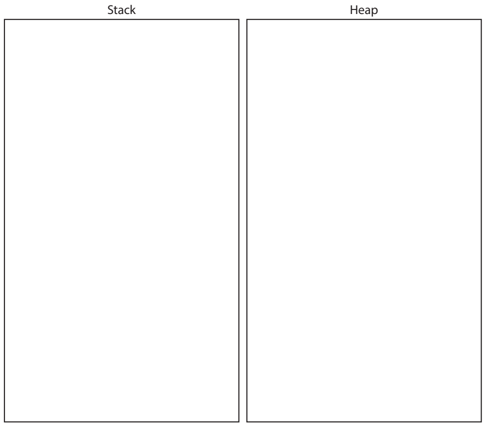
Stack vs Heap

- **Stack:** The stack is always reserved in a LIFO (last in first out) order; the most recently reserved block is always the next block to be freed.
- **Heap:** Unlike the stack, there's no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time.
- Objects created without the `new` keyword are created on the stack
- Objects created with the `new` keyword are created on the heap

Note: Objects created on stack might internally allocate memory on the heap (e.g. `std::shared_ptr`, `std::vector`, `std::string`), whereas other elements will stay completely on stack like `std::array`

Stack vs Heap

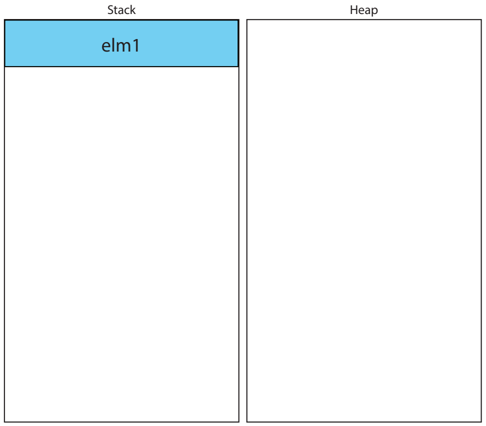
```cpp
int main() {
  bar elm1;
  bar *elm2 = new bar();
  bar *elm6;
  if( elm1.x > 0 ) {
        bar elm3;
        elm6 = foo(elm3);
  }
  delete elm2;
  delete elm6;
}
```



Stack vs Heap

```
1  int main() {
2    bar elm1;
3    bar *elm2 = new bar();
4    bar *elm6;
5    if( elm1.x > 0 ) {
6          bar elm3;
7          elm6 = foo(elm3);
8    }
9    delete elm2;
10   delete elm6;
11 }
```

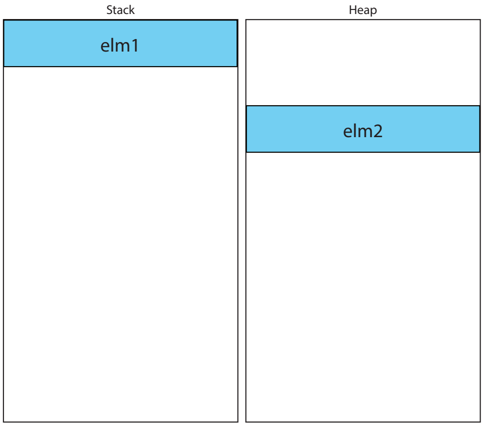
Line 2



Stack vs Heap

```
int main() {
  bar elm1;
  bar *elm2 = new bar();
  bar *elm6;
  if( elm1.x > 0 ) {
        bar elm3;
        elm6 = foo(elm3);
  }
  delete elm2;
  delete elm6;
}
```

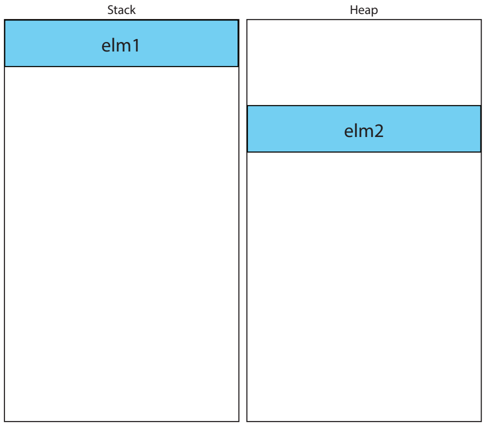
Line 3



Stack vs Heap

```
 1  int main() {
 2    bar elm1;
 3    bar *elm2 = new bar();
 4    bar *elm6;
 5    if( elm1.x > 0 ) {
 6          bar elm3;
 7          elm6 = foo(elm3);
 8    }
 9    delete elm2;
10    delete elm6;
11  }
```

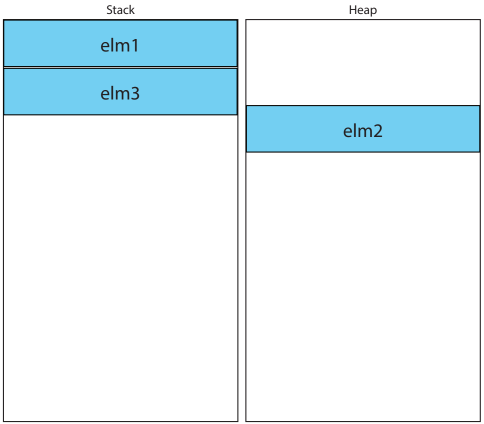
Line 4



Stack vs Heap

```
 1  int main() {
 2    bar elm1;
 3    bar *elm2 = new bar();
 4    bar *elm6;
 5    if( elm1.x > 0 ) {
 6          bar elm3;
 7          elm6 = foo(elm3);
 8    }
 9    delete elm2;
10    delete elm6;
11  }
```

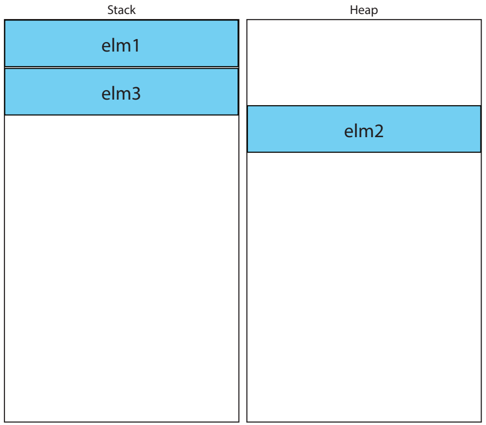
Line 6



Stack vs Heap

```
1  bar* foo( bar &elm3) {
2     bar elm4;
3     bar *elm5 = new bar();
4
5     return elm5;
6  }
```

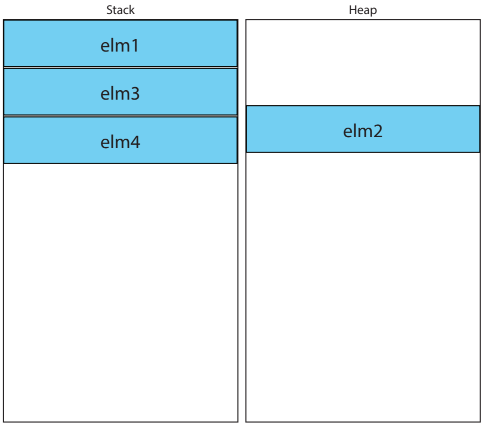
Line 1



Stack vs Heap

```
bar* foo( bar &elm3) {
   bar elm4;
   bar *elm5 = new bar();

   return elm5;
}
```

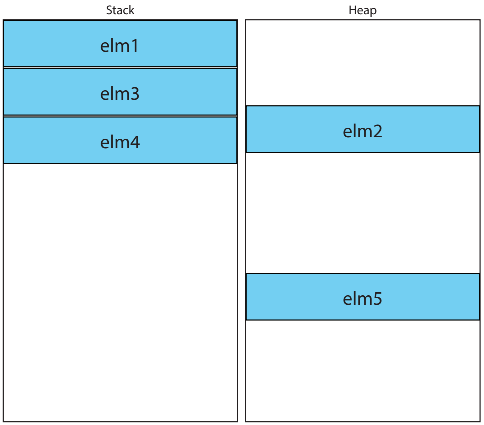
Line 2



Stack vs Heap

```
bar* foo( bar &elm3) {
   bar elm4;
   bar *elm5 = new bar();

   return elm5;
}
```

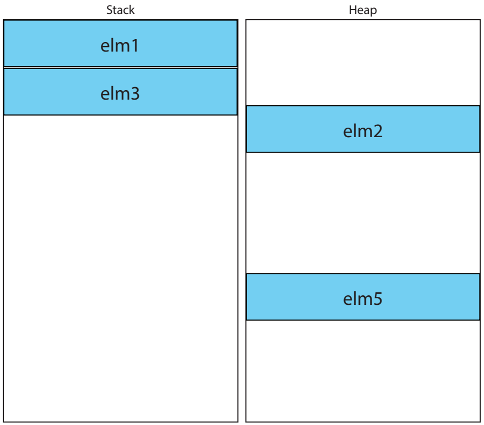
Line 3



Stack vs Heap

```
1  bar* foo( bar &elm3) {
2     bar elm4;
3     bar *elm5 = new bar();
4
5     return elm5;
6  }
```

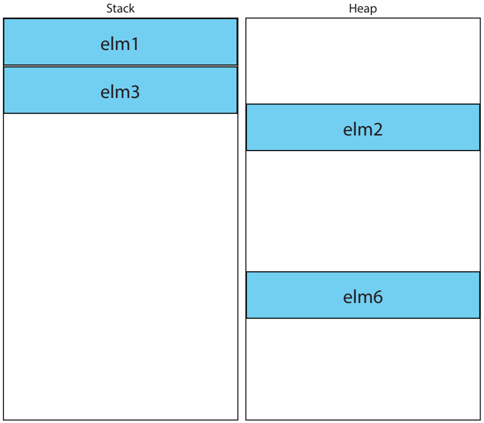
Line 6



Stack vs Heap

```
 1  int main() {
 2    bar elm1;
 3    bar *elm2 = new bar();
 4    bar *elm6;
 5    if( elm1.x > 0 ) {
 6          bar elm3;
 7          elm6 = foo(elm3);
 8    }
 9    delete elm2;
10    delete elm6;
11  }
```

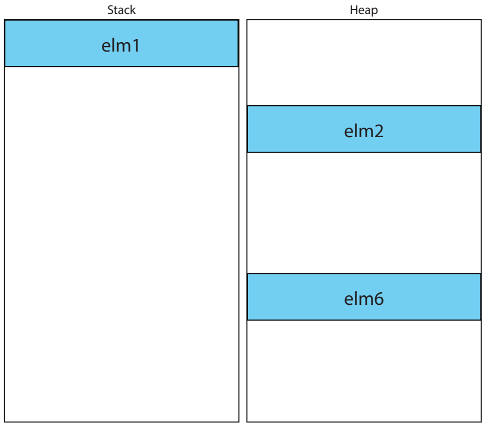
Line 7



Stack vs Heap

```
 1  int main() {
 2    bar elm1;
 3    bar *elm2 = new bar();
 4    bar *elm6;
 5    if( elm1.x > 0 ) {
 6          bar elm3;
 7          elm6 = foo(elm3);
 8    }
 9    delete elm2;
10    delete elm6;
11  }
```

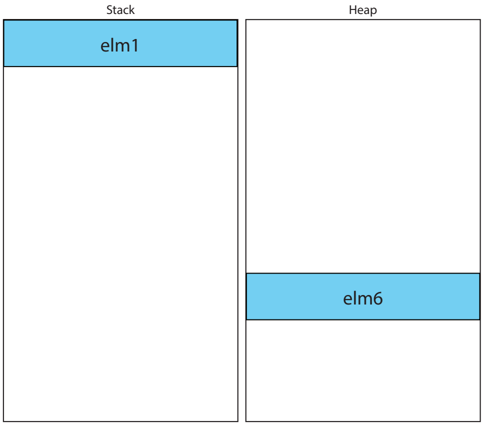
Line 8



Stack vs Heap

```
1  int main() {
2    bar elm1;
3    bar *elm2 = new bar();
4    bar *elm6;
5    if( elm1.x > 0 ) {
6          bar elm3;
7          elm6 = foo(elm3);
8    }
9    delete elm2;
10   delete elm6;
11 }
```

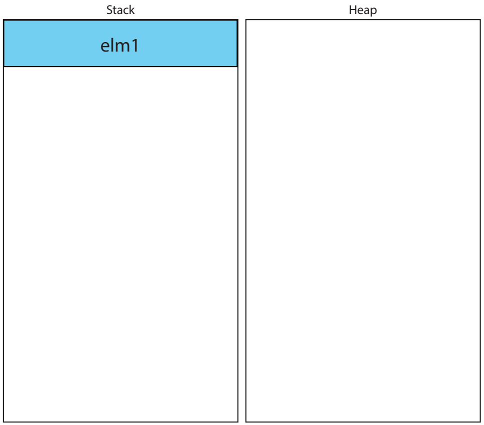
Line 9



Stack vs Heap

```
int main() {
  bar elm1;
  bar *elm2 = new bar();
  bar *elm6;
  if( elm1.x > 0 ) {
        bar elm3;
        elm6 = foo(elm3);
  }
  delete elm2;
  delete elm6;
}
```

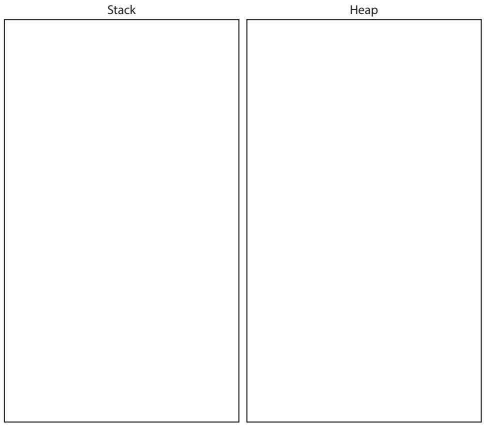
Line 10



Stack vs Heap

```
1  int main() {
2    bar elm1;
3    bar *elm2 = new bar();
4    bar *elm6;
5    if( elm1.x > 0 ) {
6          bar elm3;
7          elm6 = foo(elm3);
8    }
9    delete elm2;
10   delete elm6;
11 }
```

Line 11



Copy, Reference and Pointer

```
// pass by copy
void foo( bar elm ) {
   // elm is a copy of element in main
   elm.x = 10;
}

int main() {
  bar elm;
  elm.x = 0;

  foo(elm);
  std::cout << elm.x << std::endl; // will show 0
  return 0;
}
```

Copy, Reference and Pointer

If passed by reference, then `elm` cannot be a `nullptr`.

```
void foo( bar &elm ) {
   // elm is the same element as in 'main'
   elm.x = 10;
}

int main() {
  bar elm;
  elm.x = 0;

  foo(elm);
  std::cout << elm.x << std::endl; // will show 10

  return 0;
}
```

If an element is passed as **reference** it is not obvious for the caller that `elm` might be changed.

Copy, Reference and Pointer

Whenever possible you should pass objects by `const ref`:

```
void foo( const bar &elm ) {}
```

But this will only be possible if you do not need modify the element within the function. Therefore, you can only call member functions declared as const on a `const ref`.

Copy, Reference and Pointer

If passed by pointer, then `elm` can be a `nullptr`.

```
void foo( bar *elm ) {
   // elm points to the same element as in 'main'
   elm->x = 10;
}

int main() {
  bar elm;

  foo(&elm);
  std::cout << elm.x << std::endl; // will show 10

  return 0;
}
```

Raw pointer

```
auto *obj = new foo();
// ...
delete obj
```

Advantages:

- No internal overhead

Disadvantages:

- Object must be deleted manually, otherwise memory leaking will occur
- Pointer might point to an invalid memory address

As of C++11 you should use `std::shared_ptr` or `std::unique_ptr` instead of `new`

Shared pointer

```
auto obj = std::make_shared<foo>();

//retrieve the raw pointer of a shared pointer
auto *raw = obj.get();
```

Advantages:

- Deleting of the object is done automatically
- Pointer is either `nullptr` or is an valid object.

Disadvantages:

- If used incorrectly it might have a noticeable performance impact

Note: `std::shared_ptr` (and `std::unique_ptr`) use internally `new` to create the object, so the object will be on the heap.

Shared pointer

shared_ptr can in most situations be used like regular pointers:

```
auto obj = std::make_shared<foo>();
obj->aMethod();
```

Passing a `shared_ptr` to a function can be done in two ways.

- As a **raw pointer** to the object, if the function only uses the object for the time being called
- As a **const reference** to the shared pointer, if the function requires ownership for later use

You should pass it using a **raw pointer** if the function only uses the object for the time being called:

```
void a_function_ptr(foo *obj) {
  // uses 'obj' only in this function
}

auto obj = std::make_shared<foo>();
a_function_ptr(obj.get());
```

You should pass it using a **const reference** if the function needs to claim ownership:

```
void a_function(const std::shared_ptr<foo> &ptr) {
  // stores a copy of 'ptr' somewhere for alter use
}

auto obj = std::make_shared<foo>();
a_function(obj);
```

This will avoid unnecessary increasing and decreasing of the internal usage pointer. The **const** in this case only protects the `std::shared_ptr` from being modified, the object held by the shared ptr can still be modified.

Shared pointer

To protect the object itself from being modified, the const has to be added to the type of the std::shared_ptr:

```
void a_function(const std::shared_ptr<const foo> &ptr) {
}

auto obj = std::make_shared<foo>();
a_function(obj);
```

Weak pointer

```
std::weak_ptr<foo> weakPtr;

auto obj = std::make_shared<foo>();
weakPtr = obj;


obj = nullptr;
// weakPtr is expired at this point and will return a nullptr on lock
```

Advantages:

- Does not increase the use count, so the `weak_ptr` will not prevent an object from being deleted.

Disadvantages:

- Before the object holding by a weak pointer can be used, a `lock` has to be called, and the resulting shared_ptr has to be checked for `nullptr`.

Weak pointer

```
void a_function( const std::weak_ptr<foo> &pw ) {
  std::shared_ptr<foo> sp = pw.lock();
  if( sp != nullptr ) {
    std::cout << "pointer is valid" << std::endl;
  } else {
    std::cout << "pointer is not valid" << std::endl;
  }
}
```

It is not required or useful to check if the `weak_ptr` is expired before doing the lock, because the pointer could (only in multithreaded environments) become invalid in between those calls, so the result of `nullptr` has to be checked anyway.

Unique pointer

In contrast to std::shared_ptr only one std::unique_ptr is allowed hold a pointer to the same object.
So the following will not be allowed:

```
std::unique_ptr<foo> ptr1, ptr2;

ptr1 = std::make_unique<foo>();
ptr2 = ptr1;
```

To move an object from one unique ptr to another std::move has to be used:

```
std::unique_ptr<foo> ptr1, ptr2;

ptr1 = std::make_unique<foo>();
ptr2 = std::move(ptr1);
// at this point ptr1 does not hold the object anymore
```

Unique pointer

But it is allowed to pass it by reference, because it will still be the same `unique_ptr` object.

```
void test( const std::unique_ptr<foo> &ptr ) {}

std::unique_ptr<foo> ptr2;
ptr1 = std::make_unique<foo>();
test(ptr1);
```

But instead of passing it as reference you should just pass the raw pointer, because you don't transfer ownership in this case.

```
void test_ptr(foo *ptr ) {}

std::unique_ptr<foo> ptr2;
ptr1 = std::make_unique<foo>();
test_ptr(ptr1.get());
```

Summary

- `shared_ptr` is an owning pointer with a shared ownership. This pointer ensures that the object is released if no other shared pointer is pointing on that object anymore.
- `weak_ptr` is a pointer with a non-owning reference to an object manged by a `shared_ptr`.
- `unique_pointer` is an owing pointer with an exclusive ownership.
- Raw pointers (`observer_ptr` c++20) is a non-owning pointer. This pointer stores the address of an object, but is not responsible for the object in any way. Raw pointer can point to objects manged by a `unique_pointer` or `shared_ptr`, but won't be set to `nullptr` if those objects are deleted.

nullptr and NULL

`nullptr` was added with C++11 and replaces `NULL`.

In C++, the definition of NULL is 0, so there is only an aesthetic difference. A problem with `NULL` is that people sometimes mistakenly believe that it is different from 0 and/or not an integer. In pre-standard code, `NULL` was/is sometimes defined to something unsuitable and therefore had/has to be avoided.

The advantage of `nullptr` over `NULL` is that it is an actual type (`std::nullptr_t`) and not just an integral value.

nullptr and NULL

Test if a shared_ptr does not hold an object:

```
void test_ptr( const std::shared_ptr<foo> &ptr ) {
  if( ptr != nullptr ) {
     // has object
  } else {
     // has no object
  }
}
```

nullptr and NULL

Test if a raw pointer does not hold an object:

```
void test_ptr( foo *ptr ) {
  if( ptr != nullptr ) {
     // has object
  } else {
     // has no object
  }
}
```

nullptr and NULL

Setting a pointer to *null*:

```
std::shared_ptr<foo> ptr1 = nullptr;
foo *ptr1 = nullptr;
```

Classes and Structs

In C++ objects can be defined using `class` and `struct`:

```
class foo {
public:
  foo() {}
  ~foo() {}
protected:
   int m_somevalue;
};

struct foo {
  foo() {}
  ~foo() {}
protected:
   int m_somevalue;
};
```

Classes and Structs

The only difference between those are their default **access-specifier**.

> *In absence of an access-specifier for a base class, public is assumed when the derived class is declared struct and private is assumed when the class is declared class.*
>
> *Member of a class defined with the keyword class are private by default. Members of a class defined with the keywords struct or union are public by default.*

Classes and Structs

For a better readability of the class structure, it is useful to keep the method **declarations** and **definitions** in separate files.

- The **declarations** will be stored in a `.h` file.
- The **definitions** will be stored in a `.cpp` file.

Classes and Structs

It is required to use an include guard to prevent an endless include loop in the preprocessing step.
The classic standard conform way is:

```
#ifndef FOO_H_
#define FOO_H_

class foo {};

#endif
```

A more elegant - non standard - way that is supported by all compilers:

```
#pragma once

class foo {};
```

Classes and Structs

Class-Declaration (`.h`):

```
class foo {
public:
  foo();
  ~foo();
  int a_method(int a);
};
```

Class-Definition (`.cpp`):

```
#include "foo.h"

foo::foo() {}
foo::~foo() {}
int foo::a_method(int a) {}
```

Classes and Structs

Inheritance:

```
class polygon {
};

class triangle : public polygon {
};
```

Classes and Structs

Multiple inheritance:

```
class polygon {
};

class renderable {
};

class triangle : public polygon, public renderable {
};
```

In general it should be avoided to inherit from multiple classes.

Classes and Structs

Calling the base constructor:

```
class triangle : public polygon {
public:
    triangle();
};
```

```
#include "triangle.h"

triangle() : polygon() {}
```

The order in which the constructors are called: first base class, then inheriting class.
The destructors are called in the inverse order.

Classes and Structs

Initialization of member fields:

```
class foo {
public:
  foo(int a, int b) : m_a(a), m_b(b) {
  }

  int m_a;
  int m_b;
};
```

Classes and Structs

The destructor `~foo()` is called the moment the object is deleted. The destructor can be used to perform some cleanup tasks. Before the existence of **smart pointers** the destructor was required to free objects that where created using `new` and owned by the destructed object.

```
class foo {
public:
  foo(int a, int b) {}
  ~foo() {
        std::cout << "object is destructed" << std::endl;
  }
};
```

Classes and Structs

Visibility of member fields and methods:

- **public**: access is not restricted
- **protected**: access only within the class or in an inheriting class.
- **private**: access only within the class

The keyword **friend** allows to add exceptions, but it should be avoided to use **friend**.

Classes and Structs

Visibility of member fields and methods:

```
class foo {
public:
  foo(int a, int b) : m_a(a), m_b(b) {
  }

private:
  int m_a;
  int m_b;

protected:
  int m_c;
};
```

Classes and Structs

The keyword **virtual** has two usecase:

- **pure virtual**: to make abstract classes, requiring the inheriting class to define this method.
- **late binding**: the correct method call is determined at runtime.

Classes and Structs

Abstract class:

```
struct foo {
    virtual void test() = 0; // pure virtual
};

struct bar : public foo {
    void test() {
        std::cout << "bar::test" << std::endl;
    }
};
```

Classes and Structs

Without late binding:

```
struct foo {
    void test() {
        std::cout << "foo::test" << std::endl;
    }
};
struct bar : public foo {
    void test() {
        std::cout << "bar::test" << std::endl;
    }
};

auto b = std::make_shared<bar>();
std::shared_ptr<foo> f = b;
f->test(); // this will output "foo::test"
b->test(); // this will output "bar::test"
```

Classes and Structs

With late binding:

```
struct foo {
    virtual void test() {
        std::cout << "foo::test" << std::endl;
    }
};
struct bar : public foo {
    void test() override {
        std::cout << "bar::test" << std::endl;
    }
};

auto b = std::make_shared<bar>();
std::shared_ptr<foo> f = b;
f->test(); // this will output "bar::test"
b->test(); // this will output "bar::test"
```

Classes and Structs

Virtual destructor are important if `delete` is called on the pointer of the base class. Without **virtual** only the base destructor will be called.

```
struct foo {
    ~foo() {
        std::cout << "~foo" << std::endl;
    }
};
struct bar : public foo {
    ~bar() {
        std::cout << "~bar" << std::endl;
    }
};

foo *b = new bar();
delete foo; // this will only call the destructor of "foo"
```

Classes and Structs

With **virtual** all destructors of the object held by the Base pointer are called.

```
struct foo {
    virtual ~foo() {
        std::cout << "~foo" << std::endl;
    }
};
struct bar : public foo {
    ~bar() {
        std::cout << "~bar" << std::endl;
    }
};

foo *b = new bar();
delete b; // this will invoke the destructor of "bar" and "foo"
```

final

With the `final` keyword you can ensure that a method cannot be overwritten in a *derived* class, or that you cannot derive from a class.
A method must be virtual for it to be declared as final.

```
class foo final{
};

class bar{
    virtual void foo() final {}
};
```

Namespaces

`namespaces` are used to avoid name collisions between functions and classes/structs of different projects.
It is recommended to always place a custom function in its own namespace.
The name of the namespace can be the name of your project:

```cpp
namespace cpp_course_2017 {
  void another_function() {
  }

  void test_function() {
    another_function();
  }
}

void main() {
  cpp_course_2017::test_function();
}
```

const

The **const** keyword serves the following purposes:

- It ensures that a value cannot be changed
- A method defined as const cannot change a member of the class.
- To indicate that calling a method will not modify the object it belongs to.
- To indicate that values passed to a method/function will be unchanged

const

If a value is required to remain unchanged, then it should be defined as **const**. That way every attempt to change this value will result in a compiler error.

Without **const** it would not be easy to recognise if the value was changed by mistake in a different part of the program.

```
const int PI = 3.14159;

int main() {
  PI = 2; // compiler will report an error that PI cannot be changed
  return 0;
}
```

const

For functions/methods receiving values that will or should not be changed the **const** modifier can be used in the parameter list.

This will show the person using this function that it is save to pass a value to it without the need to worry that it might be changed.

```
void get_distance(const Point &p1, const Point &p2) {
  p1.x = 0; // will show a compiler error
}
```

const

Beside this indication for the user, it will also ensures that the value can only passed to subsequent functions that will not modify this value.

```
void set_to_zero(Point &p) {
  p.x = 0;
  p.y = 0;
}

void get_distance(const Point &p1, const Point &p2) {
  set_to_zero(p1); // compiler error
}
```

`set_to_zero` cannot be called for `p1` because `p1` is const but `set_to_zero` requires a non const reference.

const

In this example passing p1 to set_to_zero would work, because set_to_zero does not take the argument as **reference**, but will receive a copy of p1 and because of that p1 will remain unchanged.

```
void set_to_zero(Point p) {
  p.x = 0;
  p.y = 0;
}

void get_distance(const Point &p1, const Point &p2) {
  set_to_zero(p1);
}
```

const

If an object is marked as `const`, then only the methods marked as `const` can be called on this object.

```
class foo {
public:
  void testA() const;
  void testB();
};


int main() {
  const foo a;

  a.testA(); // is valid because testA is const
  a.testB(); // will not compile because testB is not const
}
```

const

Defining **const** on methods ensures that, calling such a method will not change the object or members of the object it belongs to:

```
struct Line {
   float getLength() const {
      start.x = 0; // compiler error
   }
   Point start;
   Point end;
}
```

const

The **const** on the method does not affect the values passed to this function. So this example p2 can still be modified, because the const only affect the p1 value on which copy_to is called.

```
struct Point {
  void copy_to( Point &dest ) const {
    dest.x = x;
    dest.y = y;
  }
  float x;
  float y;
};

Point p1;
Point p2;
p1.copy_to(p2);
```

const

const can also be used for member variables:

```
struct Node {
   const int someValue = 10;
};
```

const

The only place where it is allowed to change such a constant member variable is in member initializer list of the constructor.

```
struct Node {
   Node(int id) : m_id(id) {}

   const int m_id;
};
```

constexpr

The goal of `constexpr` depends on the context:

- For objects it indicates that the object is immutable and shall be constructed at compile-time.
- For functions it indicates that calling the function can result in a constant expression and can be evaluated at compile time.

In both situations it could allow certain computations to take place at compile time, literally while your code compiles rather than when the program itself is run.

constexpr

```
int get_five() {return 5;}

int some_value[get_five() + 7];
// Create an array of 12 integers. Ill-formed C++
```

```
constexpr int get_five() {return 5;}

int some_value[get_five() + 7];
// Create an array of 12 integers. Legal C++11
```

Enumeration

An enumerated data type is used whenever a set of logically related constants has to be defined.

A typical `enum` might look something like this:

```
enum Colors {
  kRed,
  kBlue,
  kGreen
};
```

Enumeration

Like in many other languages, the enums are internally represented by integral type and are by default consecutively numbered starting by 0.

```
enum Colors {
  kRed, // 0
  kBlue, // 1
  kGreen // 2
};
```

Enumeration

The numbering of the enums can be adjusted by specifying one or more initial values or by specifying all of them explicitly.

```
enum Test123 {
  a=1, // 1
  b, // 2
  c, // 3
  d=7, // 7
  e, // 8
  f // 9
};
```

Enumeration

In `C++` there are two enumeration type **unscoped enumeration** (`enum`) and since *c++11* **scoped enumeration** (`enum class`).

Usually you want to use **scoped enumeration**, because **unscoped enumeration** can result in naming conflicts and has other unexpected behaviours.

Creating a scoped enumeration is not different:

```
enum class Color {
  kRed,
  kBlue,
  kGreen
};
```

But to access the enumerator you need to use specify the name of the enum.

```
Color c = Color::kRed;
```

Enumeration

The name of an enumerator in an unscoped enumeration has to be unique within the same scope

Because of that the following code will result in a compile error:

```
enum KeyState {
  kPressed,
  kReleased
};

enum MouseButtonState {
  kPressed,
  kReleased
};
```

Enumeration

The same code will be valid using scoped enumeration:

```cpp
enum class KeyState {
  kPressed,
  kReleased
};

enum class MouseButtonState {
  kPressed,
  kReleased
};
```

Enumeration

In addition to the problem with the name conflicts, unscoped enumerations are not type-safe:

```
enum ColorA {
  red, green, blue
};

enum ColorB {
  yellow, orange, brown
};

ColorA a = yellow; // wrong enumerator assigned

if( a == red ) {
  // this will be shown in the console
  std::cout << "a is red even though yellow assigned" << std:endl;
}
```

Enumeration

In contrast to this scoped enumerations are type-safe.

```cpp
enum class ColorA {
    red, green, blue
};

enum class ColorB {
    yellow, orange, brown
};

ColorA a = ColorB::yellow; // will not compile
```

Threads

threads can be used to distribute tasks across multiple processors, allowing them to run in parallel.
Several things have to be taken into account in a multithreaded enviroment:

- Are the objects used in the thread still valid at the time they are used (whether they still exist)?
- Do the threads access the same data and if so, is this concurrent access allowed?
- The overhead of creating and managing a thread should be smaller than the performance gain you get from it (test the performance gain in release mode with code optimizations active).

Threads

When it comes to threads, there are these important classes:

- `std::thread` this class encapsulates a cross platform implementation of a thread.
- `std::async` is used to run one task async in a thread.
- `std::promise` to set a value asynchronously that can be requested in another thread
- `std::future` allows to access the result of an async task (`std::promise` and `std::async`)
- `std::mutex` is used to control concurrent access
- `std::lock_guard` is convenient method to acquire a lock on a `std::mutex`

Overloading

*C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called **function overloading** and **operator overloading** respectively.*

*An **overloaded declaration** is a declaration that had been declared with the **same name** as a previously declared declaration in the **same scope**, except that both declarations have **different arguments** and obviously **different definitions**.*

Operator overloading

Most of the build-in **operators** available in C++ can be redefined or overloaded.

Operators are functions with special names:
the keyword operator followed by the symbol for the operator.

- The operators :: (scope resolution), . (member access), .* (member access through pointer to member), and ?: (ternary conditional) cannot be overloaded.
- Operators can be overloaded as member or free function.

Operator overloading

Operation overloading can be done using member functions or free functions.

```
class vec3 {
    // for member functions op1 is the object itself
    vec3 operator + (const vec3& op2);
}

// free function
vec3 operator + (const vec3& op1, const vec3& op2);


vec3 op1, op2;
vec3 res = op1 + op2;
```

Operator overloading

- Member functions have the advantage that they can access `private` and `protected` members of op1
- Free functions have the advantage that additional operators can be added to any class, even for foreign libraries
- Using free functions the first operator does does not need to be a class type, which is not possible with member functions

```
// first operator is a float
vec3 operator + (const float& op1, const vec3& op2);
```

To define the operators consistent at the same place, the free function is often prefered over the member function.

Operator overloading

The overloads of **operator**>> and **operator**<< that take a `std::istream&` or `std::ostream&` as the left hand argument are known as insertion and extraction operators. Since they take the user-defined type as the right argument, they must be implemented as non-members.

```
std::ostream& operator<<(std::ostream& os, const T& obj)
{ // write obj to stream
    return os;
}
std::istream& operator>>(std::istream& is, T& obj)
{ // read obj from stream
    return is;
}
```

If they are not declared as friend function, only public members and methods of T can be accessed.

Function overloading

- An overloaded function has multiple definitions for the same function name in the same scope.
- They differ from each other by the types and/or the number of arguments.
- It is not possible, to overload function declaration by just changing a parameter type to const.
- **Changing a reference or pointer parameter to const** will overload a function. *Thus, it is possible to overload function declarations with const keyword, which changes the implicitly given self-reference.*
- **You cannot overload function declarations that differ only by return type.**

Function overloading

```
void foo(int i) {}
void foo(const int i) {} //function has already been defined
int main(){

}
```

```
void foo(int i){}
void foo(int& i){}

int main(){
       foo(0); //ambiguous call to overloaded function
}
```

Function overloading

```
void foo(int& i){
        std::cout << "non-const" << std::endl;
}

void foo(const int& i){
        std::cout << "const" << std::endl;
}

int main() {
        int i1 = 0;
        foo(i1); // prints non-const
        const int i2 = 0;
        foo(i2); // prints const
}
```

Function overloading

```
void foo(int* pi){
        std::cout << "non-const" << std::endl;
}

void foo(const int* pi){
        std::cout << "const" << std::endl;
}

int main() {
        int i1 = 0;
        foo(&i1); // prints non-const
        const int i2 = 0;
        foo(&i2); // prints const
}
```

Function overloading

Note: `int* pi` and `int* const pi` cannot be used for function overloading as they both point to a mutable int.

```
void foo(int* pi) {}
void foo(int* const pi) {} //function has already been defined

int main(){

}
```

Function overloading

```
class bar
{
  public:
        void foo() {};
        void foo() const {};
}
```

Possible overloading because the self-reference (implicit first function parameter) changes its type from `bar&` to `const bar&`.

Templates

templates allow to create helper/utility functions or classes without the need to know the type they are used with.
Examples for **template** based functions are:

- Comparison functions like `max`, `min`, `accumulate`, ...
- Containers of the standard library like `vector`, `list`, ...
- Helper functions like `sort`, `find`, ...

The compiler will check at compile time if the data types passed to the template based functions or classes can be used with those.

Templates

Advantages:

- Only one function has to be created and can be used with all data types that meet the conditions of the template function.
- Certain tasks can be performed that would otherwise be only possible if inheritance is used.
- Dependencies between in the code can be reduced.

Disadvantages:

- Compiler errors are not easy to understand.
- Compiler errors that might exist if used with a certain data type will only occur if those data type are used.
- Writing template base functions/classes is not as intuitive as regular functions.

Templates

Without **templates** a function like `std::max` has to be created for each type the function should be used with.
A function for `int` would look that way:

```
int max( const int &a, const int &b ) {
  if( a > b ) return a;
  else return b;
}
```

Templates

If `max` should also support `float` an additional **overloading** function has to be created for `float`:

```
int max( const int &a, const int &b ) {
  if( a > b ) return a;
  else return b;
}

float max( const float &a, const float &b ) {
  if( a > b ) return a;
  else return b;
}
```

Templates

This has do be done for every data type that `max` should support and will result in a huge number of nearly identical functions that only differ in their type. And it will work only for data types that are know at the time the code was written.

To prevent the need to create a separate overloading function for each data type **templates** are used.

Templates

The template based version of **max** is:

```
template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}
```

The `template<typename T>` shows that the function has one type that is resolved at compile time. The `T` is the *placeholder* for this data type. Because `T` is used for both `a` and `b` the data type of the values passed as argument to `max` has to be the same.

This function does accept every data-type that can be compared using the > operator.

Sometimes you might see `class` instead of `typename`. The keyword `class` exists only for backwards compatibility, but has the same meaning.

Templates

The compiler will now take over the task to create those overloaded functions as soon as they are needed.

```
template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}

float x_f = 0.1f, y_f = 0.2f;
int x_i = 10, y_i = 20;

auto max_f = max(x_f, y_f); // returns 0.2f
auto max_i = max(x_i, y_i); // returns 20
```

Templates

If the data-type passed to that function are different, the compiler will show an error message like `No matching function call for 'max'`.

```
template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}

float x_f = 0.1f;
int y_i = 10;

auto max_f = max(x_f, y_i); // will result in a compiling error
```

Templates

The data-type used for `T` can be set explicitly to force the use of a certain version of the template function.

```
template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}

float x_f = 20.1f;
int y_i = 10;

auto max_f = max<float>(x_f, y_i); // returns 20.1f
```

Templates

The data-type used for `T` can be set explicitly to force the use of a certain version of the template function.

```
template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}

float x_f = 20.1f;
int y_i = 10;

auto max_i = max<int>(x_f, y_i); // returns 20 because x_f is casted to int
```

Templates

Passing a data-type that is not comparable using > to this function will result in a compiler error at the comparison. In this example the error would be something like `Invalid operands to binary expression`. The error message depends on the kind of compiler error that would occur for this data-type with the given operation.

```
class foo {};

template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a; // compiling fails at this point
  else return b;
}

foo x, y;

auto max_i = max(x, y); // error might also reported for this line
```

Templates

If the data-type is passed to another function the error could be something like `No matching function call to 'sqrt'`.

```cpp
class foo {};

template<typename T>
T sqrt_minus_one( const T &a ) {
  return std::sqrt(a) - 1; // compiling fails at this point
}

foo x;
sqrt_minus_one(x);
```

Templates

Templates can not only be used for simple data-types like int, float, ... but also for class and struct. To print out the content of std::list and std::vector or any other data type that supports iterators one template function can be created.

```
template<typename T>
void print_container_content( const T &container ) {
  for( auto const &item : container ) {
    std::cout << item << std::endl;
  }
}

std::vector<float> v = {1,2,3,4};
std::list<float> l = {1.1f,2.1f,3.1f,4.1f};
print_container_content(v);
print_container_content(l);
```

Templates

This example also shows another useful case for the `auto` keyword. Because the function would accept a container with any data-type the data-type of `item` depends on the container that is passed.

```
template<typename T>
void print_container_content( const T &container ) {
  for( auto const &item : container ) {
    std::cout << item << std::endl;
  }
}

std::vector<float> v = {1,2,3,4};
std::list<float> l = {1.1f,2.1f,3.1f,4.1f};
print_container_content(v);
print_container_content(l);
```

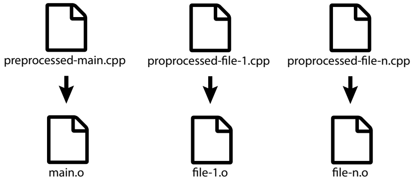
Templates

Without the auto keyword it would be required that the passed container has a static field that exposes its type. Standard conform containers do this with the value_type field. To be able to use this data-type in the code typedef typename T::value_type has to be used to give it a name that then can be used.

```
template<typename T>
void print_container_content( const T &container ) {
    typedef typename T::value_type ValueType;

    for( ValueType const &item : container ) {
        std::cout << item << std::endl;
    }
}
```

Templates

Because the compiler generates the code for a template based function only if it is used, the **definition** of the template function has to be *visible* for each compilation unit in which this function is used.

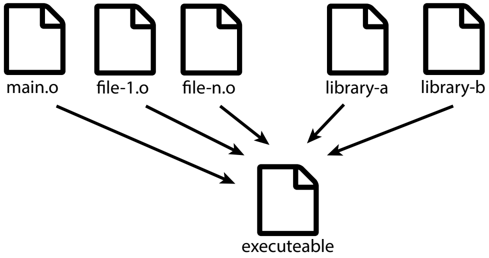


For this reason it is not possible to write the **definition** of a template function into a separate `.cpp` file, but it must be written into a `.h` write.
This header file must then be included in the `.cpp` file where the template function is used, otherwise the linker will report an error.

Templates

If two .cpp use the same template function with the same data-types as argument, then the same generated function would exist in multiple **translation units**.

And because the linker will not only check if all **definitions** for each **declaration** exist, but also ensures that there are no duplicate **declaration** in each of the **translation units**, the linking would fail.



Inline

To solve this problem those functions have to be defined as **inline**.

Meaning of the keyword **inline** is often mistakenly explained as feature to allow compilers to optimise code: that would insert the code of the function directly at the place where the function is called, removing the function call itself.

The meaning of the keyword **inline** for functions is **multiple definitions are permitted** rather than **inlining is preferred**. Compilers are able to decide by themselves if such optimisations are possible and useful.

A function defined entirely inside a class/struct/union definition, whether it is a member function or a non-member friend function, is implicitly an inline function.

Inline

inline can also be used for non-template functions and methods if you want to place them in the header.

```cpp
void foo();
// ... more declarations

// followed by their definitions in the same header
inline void foo() { }
```

```cpp
class foo {
  void test();
  // ... more declarations
};

// followed by their definitions in the same header
inline void foo::test() { }
```

Templates

A function can be defined as **inline** by adding the `inline` keyword in front of it.

```
template<typename T>
inline void print_container_content( const T &container ) {
    for( auto const &item : container ) {
        std::cout << item << std::endl;
    }
}
```

Inline can also be used with non-template functions if you want to write the declaration in the header.

Templates

If it is required to use the template function with a data-type that is not compatible with the generic version of the template function, then it is possible to specialize the template for a certain data-type.
This template function would not work with `foo` because it is not comparable using the > operator.

```
struct foo {
  int m_value;
};

template<typename T>
T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}
```

Templates

A specialized version of the max function can be created for this data-type.

```
template<typename T>
inline T max( const T &a, const T &b ) {
  if( a > b ) return a;
  else return b;
}

template<>
inline foo max<foo>( const foo &a, const foo &b ) {
  if( a.m_value > b.m_value ) return a;
  else return b;
}
```

Templates

Template functions can be useful to solve certain problems without the need of inheritance.

If a function is needed that accepts both `DataTypeA` and `DataTypeB` and this function has to call `aFunc`.

```
struct DataTypeA {
  void aFunc() {}
}
struct DataTypeB {
  void aFunc() {}
}
```

Templates

Then inheritance and **virtual** could be used:

```
struct Base {
  virtual void aFunc();
};
struct DataTypeA : public Base {
  void aFunc() {}
};
struct DataTypeB : public Base {
  void aFunc() {}
};

void do_something( const std::shared_ptr<Base> &obj ) {
        obj->aFunc();
}
```

Templates

Or you could create a template based function:

```
struct DataTypeA {
  void aFunc() {}
};
struct DataTypeB {
  void aFunc() {}
};
template<typename T>
void do_something( const std::shared_ptr<T> &obj) {
  obj->aFunc();
}
```

Templates

Templates cannot only be used with functions but also with structs and classes.

```
template<typename T>
struct Point {
    T x;
    T y;

    Point(T x, T y) : x(x), y(y) {}

    void add( const Point<T> &pt ) {
        x = x + pt.x;
        y = y + pt.y;
    }
};
```

Templates

And also for methods inside of the class/struct:

```
template<typename T>
struct Point {
    // ...
    void add( const Point<T> &pt ) {
        // ...
    }

    template<typename aT>
    void add( const Point<aT> &pt ) {
        x = x + pt.x;
        y = y + pt.y;
    }
};
```

Default parameters

A default parameter (also called an optional parameter or a default argument) is a function parameter that has a default value provided to it.

If a value for a parameter is not supplied by the user, the default value will be used.

```cpp
void printValues(int x, int y = 10, int z = 20){
        std::cout << "x: " << x << " y: " << y << " z: " << z << std::endl;
}

int main(){
        printValues(1); // prints x: 1 y: 10 z: 20
        printValues(3, 4); // prints x: 3 y: 4 z: 20
        printValues(7, 8, 9); // prints x: 7 y: 8 z: 9
}
```

Multiple default parameters are allowed, but all default parameters must be the rightmost parameters.

Default parameters

Multiple default parameters are allowed, but all default parameters must be the rightmost parameters.

```
void printValue(int x = 10, int y); // not allowed
void printValue(int x = 10, int y, int z = 20); // not allowed
```

Default paramters can only be declared once, either in the function declaration or function definition.

```
void printValues(int x, int y = 10);

void printValues(int x, int y = 10) {
// error: redefinition of default parameter
       std::cout << "x: " << x << " y: " << y << std::endl;
}
```

Default parameters

The effect of default parameters can be achieved by overloading:

```
void foo(int a, int b = 0);
```

is equivalent to

```
void foo(int a, int b);
void foo(int a) {
       foo(a, 0);
}
```

This approach cannot be used for constructors!

std::function

`std::function` is a wrapper for functions. It allows to store a reference to a function that can be called later.

Those function objects can i.e. be used to pass a custom comparison to a function.

As template parameter the `std::function` takes the *function signature*, a `std::function` that should be able to wrap this function:

```cpp
bool is_greater(int a, int b) {
  return a > b;
}
```

Has to be defined as `std::function<bool(int,int)>`.

std::function

A function that would accept a different function with the signature `bool(int,int)` as argument would look like this:

```cpp
bool does_accept_value(int a, int b,
                       const std::function<bool(int,int)> &condition) {
  return condition(a, b);
}
```

std::function

```cpp
bool is_greater(int a, int b) {
  return a > b;
}

bool is_less(int a, int b) {
  return a > b;
}

bool does_accept_value(int a, int b,
                       const std::function<bool(int,int)> &condition) {
  return condition(a, b);
}

does_accept_value(1, 3, is_greater);
does_accept_value(1, 3, is_less);
```

std::function

A `std::function` object could be stored in a member field for later use:

```
struct Task {
   void callWhenFinished(const std::function<void<Task*>> &callback) {
     m_callWhenFinished = callback;
   }
   std::function<void<Task*>> m_callWhenFinished;
}

void task_finished(Task * task) {
   std::cout << "task finished" << std::endl;
}

Task task;
task.callWhenFinished(task_finished);
```

std::function

`std::function` are used by utility functions like `sort`, `find_if` to allow custom sort or find conditions.

```cpp
bool is_even(int val) {
  return ((val % 2) == 0);
}

std::vector<int> values = {1,2,3,4};

auto it = std::find_if(values.begin(), values.begin(), is_even);
```

std::function

Before `std::function` was introduced this had to be done using function pointers.

```
void testA(int(*callback)(int)) {
  std::cout << callback(2) << std::endl;
}

int test2(int val) {
  return val * 2;
}

int main() {
  testA(&test2);
}
```

Nowadays this is only needed if you work with old code or when interacting with libraries that only have a C interface.

std::function

The same code using `std::function`:

```
void testA(const std::function<int(int)> &callback) {
  std::cout << callback(4) << std::endl;
}

int test2(int val) {
  return val * 2;
}

int main() {
  testA(test2);
}
```

Using a `std::function`, it is easier to recognize the return type and arguments of the function, and how the function pointer is called.

Lambda expressions

A Lambda expression constructs an unnamed function object capable of capturing variables in scope (a closure). It allows to use functions that accept a `std::function` with out the need to define a function.
This is useful if the passed function is required only at one specific place and would allow to change:

```
bool is_greater(int a, int b) {
  return a > b;
}

bool does_accept_value(int a, int b,
                        const std::function<bool(int,int)> &condition) {
  return condition(a, b);
}

does_accept_value(1, 3, is_greater);
```

Lambda expressions

To this:

```
bool does_accept_value(int a, int b,
                       const std::function<bool(int,int)> &condition) {
  return condition(a, b);
}

does_accept_value(1, 3, [](int a, int b) -> bool {
  return a > b;
});
```

Lambda expressions

The Lamda expression starts with an [] and the return type is defined after the parameters using ->, beside that it looks like a regular function.

```
[](int a, int b) -> bool {
  return a > b;
}
```

Lambda expressions

The [] in front of the lambda expression denote how variables that are defined in the function in which the lambda expression is created, are passed to the lambda function. If [] is empty, then not variables are passed:

```
int main() {
  Point p;
  auto fun = []() -> void {
    p.x = 0; // compiler error
  }
}
```

p is not accessible within the lambda function.

Lambda expressions

A `&` within the `[]` instructs the compiler to pass all values defined in `main` as reference to the lambda.

```cpp
int main() {
  Point p;
  p.x = 1;
  auto fun = [&]() -> void {
    p.x = 20;
  }
  fun();
  std::cout << p.x << std::endl; // will show 20
}
```

Now `p` is accessible within the lambda and refers to the same object.

Lambda expressions

A = within the [] instructs the compiler to pass all values defined `main` as copy to the lambda.

```
int main() {
  Point p;
  p.x = 1;
  auto fun = [=]() -> void {
    p.x = 20;
  }
  fun();
  std::cout << p.x << std::endl; // will show 1
}
```

Now `p` is accessible within the lambda, but refers to a copy of the `p` in main.

Lambda expressions

The settings within the [] can be adapted for individual variables.

```
int main() {
  Point p1;
  Point p2;

  auto fun = [p1,&p2]() -> void {

  };
  fun();
}
```

Now p1 is passed as copy and p2 by reference.

Lambda expressions

A lamdba function can be invoked directly without the need of storing it:

```cpp
int main() {
  auto val = []() -> int {
    return 3;
  }();
  std::cout << val << std::endl; // will show 3
}
```

Lambda expressions

Lambda expressions can be used with utility functions like `sort`, `find_if` that allow custom sort or find conditions.

```
std::vector<int> values = {1,2,3,4};

auto it = std::find_if(values.begin(), values.begin(),
  [](int val) -> bool {
        return ((val % 2) == 0);
});
```

Lambda expressions

If the compiler can determine type of the arguments then `auto` can be used for the argument, and the return type can be omitted.

```
std::vector<int> values = {1,2,3,4};

auto it = std::find_if(values.begin(), values.begin(),
                        [](auto val) {
                            return ((val % 2) == 0);
                        });
```

assert

There are two ways of error checking:

- at compile-time using `static_assert`
- or at runtime, with i.e. `assert` that is defined in the `cassert` header.

assert

The assert in the cassert is a functionality that allows to do runtime-checks that are only performed if the code is compiled in debug mode. The purpose is to create checks for situations that should never happen.

```
void divide_by( float a, float b ) {
  assert(b!=0); // will throw an error if (b!=0) is not true
}

divide_by(1.f, 0.f);
```

Using assert(b!=0) here is only valid if you expect that it should never be possible to call divide_by with b having the value 0.

assert

If it might be possible that b could be 0 in certain situations, then you should use exceptions instead:

```
void divide_by( float a, float b ) {
  if( b == 0 ) {
     throw new std::runtime_error("divide by zero not allowed");
  }
}
```

And at the place where calling divide_by could pass 0 for the parameter b then should take care about the exception.

assert

Everting that is inside of braces of `assert()` will not be executed if the code is not build in debug mode. So you should never manipulate an object or a value within the `assert`:

```
int i=1;
assert(i++>0)
assert(i++>0)
std::cout << i << std::endl;
```

This will show 1 as output when build in release mode and 3 when build in debug mode.

assert

`static_assert` allows to do checks at compile time:

- Check if resolved types for template parameters full-fill certain conditions
- Check if the data-types of the platform for which the the code is compiled meets certain conditions
- Check if constant values have an expected value

assert

To ensure that a template based function can only be used with integral data-types, `static_assert` can be used in combination with `std::is_integral`

```
template <class T>
T sum_values(T a, T b)
{
   static_assert(std::is_integral<T>::value,
                 "T has to be an integral type (short, int, long, ...)");

}
```

assert

If the code would only run reliable if the size of `int` is 4 byte, then a global `static_assert` that checks this condition can be added:

```
static_assert(sizeof(int) == 4, "size of int must be 4");


int main() {
}
```

The code will now fail to compile if the `int` has a different size on the platform it is compiled for.

assert

If a library exposes the version of its API as a constant variable in the header of this library, then `static_assert` assert could be used to ensure that the code will always be compiled against a compatible API version:

```
static_assert(sizeof(a_library::major_version) >= 2,
  "The version of library 'a_library' has to be at least 2.0");

int main() {
}
```

[[deprecated]]

In an evolving project it might happen that certain functions do not meet the overall requirements anymore, are replaced by different functions, and should not be use any longer because of that.

If those functions are used only a few times through out the code, then it is not a problem to replace them directly by their successors.

If they are used at many places or if other projects depend on this code then it becomes a time-consuming and confusing task to replace all those functions. Because you would need to continuously do manual textual searches through the whole code to look for all calls to those functions until all of them are replaced.

To overcome this problem the attribute specifier sequence `[[deprecated]]` was introduced with C++14.

[[deprecated]]

To mark a function/method as deprecated the `[[deprecated]]` attribute specifier can be added right before the function definition:

```
[[deprecated]]
void do_some_calcualtions(const std::vector<float> &list);
```

The compiler will now report a `do_some_calcualtions is deprecated` for every place where `do_some_calcualtions` is still used. So a manual search is not required anymore.

[[deprecated]]

The `[[deprecated("reason")]]` attribute specifier allows to define an additional message that will be displayed with the compiler waring.

This is useful if the functionality of the deprecated function is replaced by one or more alternative functions. Using the Additional message allows to provide the information what functions should be used as replacement with the compiler warning itself.

```
[[deprecated("use 'prepare_data' and 'do_calcualtion' instead")]]
void do_some_calcualtions(const std::vector<float> &list) {
       prepare_data(list);
       do_calcualtion(list);
}
```

The compiler will now report a `do_some_calcualtions is deprecated: use 'prepare_data' and 'do_calcualtion' instead`.

static

Meanings of the static keyword:

- internal linkage
- static local variables
- static class members

Storage duration

All objects in a program have one of the following storage durations:

- **automatic** storage duration. The object is allocated at the beginning of the enclosing code block and deallocated at the end. All local objects have this storage duration, except those declared `static`, `extern` or `thread_local`.
- **static** storage duration. The storage for the object is allocated when the program begins and deallocated when the program ends. Only one instance of the object exists. All objects declared at namespace scope (including global namespace) have this storage duration, plus those declared with `static` or `extern`.
- **dynamic** storage duration. The object is allocated and deallocated per request by using dynamic memory allocation functions (e.g. smart pointers).
- **thread** storage duration. The object is allocated when the thread begins and deallocated when the thread ends. Each thread has its own instance of the object. Only objects declared `thread_local` have this storage duration. `thread_local` can appear together with `static` or `extern` to adjust linkage.

static: internal linkage

```
static int i = 42;
void doSomething(){
    cout << i;
}
```

In this example, **static** tells the compiler to make the variable **i** available only in the current translation unit (file). If another file tries to use **i**, perhaps using the `extern` keyword, it would generate a linker error (unresolved external symbol).
The same concept applies to functions whose access you want to limit to the current file. This way, you can name variables/functions whatever you like without risking a naming collision in the global namespace.

static: local variables

```
void foo(){
    static int callCount = 0;
    callCount++;
    cout << "foo has been called " << callCount << " times" << endl;
}
```

Variables declared at block scope with the specifier static have static storage duration but are initialized the first time control passes through their declaration (unless their initialization is zero- or constant-initialization, which can be performed before the block is first entered). On all further calls, the declaration is skipped.

static: class members

```
//in .h file
class MyClass{
    public:
        static int FavoriteNumber; //cannot be initialized here
        const static int FavoriteNumer2 = 42; // OK
        static void foo();
};
//in .cpp file
int MyClass::FavoriteNumber = 42;

void MyClass::foo(){
    // can't access member variables here!
}
```

`foo()` can be called without object just with `MyClass::foo()`.

Compiler flags

Invoke GNU C++ compiler from command line:

```
$g++ -o helloworld helloworld.cpp
```

With compiler flags:

```
$g++ -Werror -pedantic -MoreFlags -o helloworld helloworld.cpp
```

Compiler flags

With CMake:

```
cmake_minimum_required(VERSION 3.7)
project(HelloWorld)

set(CMAKE_CXX_STANDARD 14)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -pedantic -Wextra")

set(SOURCE_FILES helloworld.cpp)
add_executable(HelloWorld ${SOURCE_FILES})
```

Compiler flags

Some compiler flags for warnings (GNU compiler):

- -**Wall**: This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning), even in conjunction with macros.
- -**Werror**: Make all warnings into errors.
- -**pedantic**: Issue all the warnings demanded by strict ISO C and ISO C++ .
- -**Wextra**: Enables some extra warning flags that are not enabled by -Wall.

Note: *-std=XXX* changes the C++ version used (default depends on compiler version).

Includes and Libraries

Additional include folders and external libraries can be added with `CMake`.

- Includes: `target_include_directories(<target> directory1 directory2 ...)`
- Libraries: `target_include_directories(<target> Library1 Library2 ...)`

Build `<target>` has to be defined first, e.g. with `add_executable` or `add_library`.

Includes and Libraries

```cmake
cmake_minimum_required(VERSION 3.7)
project(HelloWorld)

set(CMAKE_CXX_STANDARD 14)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -pedantic -Wextra")

set(SOURCE_FILES helloworld.cpp)
add_executable(HelloWorld ${SOURCE_FILES})

#after add_executable
target_include_directories(HelloWorld /path/to/my/directory)
target_link_libraries(HelloWorld Library1 Library2 ...)
```

CMake: find_package

Finds and loads settings from an external project.

```
    find_package(<package> [version]
        [EXACT][QUIET][REQUIRED][COMPONENTS] ...)
```

If found, the following variables will be defined:

- `<package>_FOUND`
- `<package>_INCLUDE_DIRS` or `<package>_INCLUDES`
- `<package>_LIBRARIES` or `<package>_LIBS`
- `<package>_DEFINITIONS`

CMake comes with numerous modules that aid in finding various well-known libraries and packages. You can get a listing of which modules your version of CMake supports by typing `cmake -help-module-list`, or by figuring out where your modules-path is (e.g. /usr/share/cmake/Modules/).
Modules follow the naming convention `Find<package>.cmake`.
Missing modules are often available online or can be written easily.