Cpp concept project

Generated by Doxygen 1.8.20

Chapter 1

C++ concepts project

See the `Documentation`!

1.1 Idea

This project serves as sample/concept project for further projects :thumbsup:

1.2 Related documents

- `Notes`
- Markdown cheatsheet
- Project structure
- Unit testing

1.3 Structure

1.3.1 Folders

- **bin**: output executables go here (for the app, tests and spikes)
- **build**: containing all the object files (removed by clean)
- **doc**: documentation files
- **ideas**: smaller classes or files to test technologies or ideas
- **include**: all project header files, all necessary third-party header files (which are not in /usr/local/include)
- **lib**: any library that get compiled by the project, third party or any needed in development
- **resources**: resources
- **src**: the application and application's source files
- **test**: all test code files

1.4 Content (Concepts)

1.4.1 Programming concepts

- Classes
  - Inheritance
- Templates
- ...

1.4.2 Documentation

The documentation is intrinsically implemented using `doxygen`. In order to do that:

- specify path to doxygen binary in the Makefile
- execute *make doc*

The *README.md* file is used for the Mainpage of the documentation. Set the settings for doxygen in *doc/Doxyfile*.

1.4.3 Makefile

Following targets are implemented:

- **all** default make
- **remake**
- **clean**
- **cleaner**
- **resources**
- **sources**
- **directories**
- **ideas**
- **tester**
- **doc**

Chapter 2

CMake

2.1 Links

- Repository
- Awesome-CMake list

2.1.1 Documentation

- CMake official documentation
- The Architecture of Open Source Applications

2.1.2 Tutorials & Instructions

- Effective Modern CMake (Dos & Don'ts)
- GitBook: Introduction to Modern CMake
- CMake Cookbook
- CMake Primer

2.1.3 Videos

- Intro to CMake
- Using Modern CMake Patterns to Enforce a Good Modular Design
- Effective CMake
- Embracing Modern CMake

2.2 Basics

2.2.1 CMake Version

```
#minmum CMake version
cmake_minimum_required(VERSION 3.12)
if(${CMAKE_VERSION} VERSION_LESS 3.12)
    cmake_policy(VERSION ${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
endif()
```

2.2.2 VARIABLES

```
# Local variable
set(MY_VARIABLE "value")
set(MY_LIST "one" "two")
# Cache variable
set(MY_CACHE_VARIABLE "VALUE" CACHE STRING "Description")
# Environmental variables
set(ENV{variable_name} value) #access via $ENV{variable_name}
```

2.2.3 PROPERTIES

```
set_property(TARGET TargetName PROPERTY CXX_STANDARD 11)
set_target_properties(TargetName PROPERTIES CXX_STANDARD 11)
get_property(ResultVariable TARGET TargetName PROPERTY CXX_STANDARD)
```

2.2.4 Output folders

```
# set output folders
set(PROJECT_SOURCE_DIR)
set(CMAKE_SOURCE_DIR ...)
set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}$/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})
```

2.2.5 Sources

```
# set sources
set(SOURCES example.cu)
file(GLOB SOURCES *.cu)
```

2.2.6 Executables & targets

Add executable/create target:

```
#add_executable(example ${PROJECT_SOURCE_DIR}/example.cu)
add_executable(miluphcuda ${SOURCES})
# add include directory to target
target_include_directories(miluphcdua PUBLIC include) #PUBLIC/PRIVATE/INTERFACE
# add compile feature to target
target_compile_features(miluphcuda PUBLIC cxx_std_11)
# chain targets (assume "another" is a target)
add_library(another STATIC another.cpp another.h)
target_link_libraries(another PUBLIC miluphcuda)
```

2.2.7 PROGRAMMING IN CMAKE

Keywords:

- NOT
- TARGET
- EXISTS
- DEFINED
- STREQUAL
- AND
- OR
- MATCHES
- ...

2.2.7.1 Control flow

```
if(variable)
    # If variable is `ON`, `YES`, `TRUE`, `Y`, or non zero number
else()
    # If variable is `0`, `OFF`, `NO`, `FALSE`, `N`, `IGNORE`, `NOTFOUND`, `""`, or ends in `-NOTFOUND`
#endif()
```

2.2.7.2 Loops

- `foreach(var IN ITEMS foo bar baz) ...`
- `foreach(var IN LISTS my_list) ...`
- `` `foreach(var IN LISTS my_list ITEMS foo bar baz) ... ``

2.2.7.3 Generator expression

```
target_include_directories(
        MyTarget
        PUBLIC
        $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
        $<INSTALL_INTERFACE:include>
)
```

2.2.7.4 Functions (& macros)

```
function(SIMPLE REQUIRED_ARG)
    message(STATUS "Simple arguments: ${REQUIRED_ARG}, followed by ${ARGV}")
    set(${REQUIRED_ARG} "From SIMPLE" PARENT_SCOPE)
endfunction()
simple(This)
message("Output: ${This}")
```

2.2.8 COMMUNICATION WITH CODE

2.2.8.1 Configure File

```
configure_file()
...
```

2.2.8.2 Reading files

...

2.2.9 RUNNING OTHER PROGRAMS

2.2.9.1 command at configure time

```
find_package(Git QUIET)
if(GIT_FOUND AND EXISTS "${PROJECT_SOURCE_DIR}/.git")
    execute_process(COMMAND ${GIT_EXECUTABLE} submodule update --init --recursive
            WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
            RESULT_VARIABLE GIT_SUBMOD_RESULT)
    if(NOT GIT_SUBMOD_RESULT EQUAL "0")
        message(FATAL_ERROR "git submodule update --init failed with ${GIT_SUBMOD_RESULT}, please checkout
       submodules")
    endif()
endif()
```

2.2.9.2 command at build time

```
find_package(PythonInterp REQUIRED)
add_custom_command(OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp"
        COMMAND "${PYTHON_EXECUTABLE}" "${CMAKE_CURRENT_SOURCE_DIR}/scripts/GenerateHeader.py" --argument
        DEPENDS some_target)
add_custom_target(generate_header ALL
        DEPENDS "${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp")
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp DESTINATION include)
```

2.3 Libraries

```
# make a library
add_library(one STATIC two.cpp three.h) # STATIC/SHARED/MODULE
```

2.4 Language/Package related

2.4.1 C

```
# set C compiler
set(CMAKE_C_COMPILER "/usr/bin/gcc-7")
execute_process (
        COMMAND bash -c "git describe --abbrev=4 --dirty --always --tags'"
        OUTPUT_VARIABLE GIT_VERSION
)
#set(CMAKE_BUILD_TYPE DEBUG)
set(CMAKE_C_FLAGS "-c -std=c99 -O3 -DVERSION=\"${GIT_VERSION})\" -fPIC")
#set(CMAKE_C_FLAGS_DEBUG "-O0 -ggdb")
#set(CMAKE_C_FLAGS_RELEASE "-O0 -ggdb")
```

2.4.2 C++

...

2.4.3 CUDA

See Combining CUDA and Modern CMake

2.4.3.1 Enable Cuda support

CUDA is not optional
```
project(MY_PROJECT LANGUAGES CUDA CXX)
```

CUDA is optional
```
enable_language(CUDA)
```

Check whether CUDA is available
```
include(CheckLanguage)
check_language(CUDA)
```

2.4.3.2 CUDA Variables

Exchange *CXX* with *CUDA*

E.g. setting CUDA standard:
```
if(NOT DEFINED CMAKE_CUDA_STANDARD)
    set(CMAKE_CUDA_STANDARD 11)
    set(CMAKE_CUDA_STANDARD_REQUIRED ON)
endif()
```

2.4.3.3 Adding libraries / executables

As long as ∗.cu∗ is used for CUDA files, the procedure is as normal.

With separable compilation
```
set_target_properties(mylib PROPERTIES
                            CUDA_SEPARABLE_COMPILATION ON)
```

2.4.3.4 Architecture

Use `CMAKE_CUDA_ARCHITECTURES` variable and the `CUDA_ARCHITECTURES property` on targets.

2.4.3.5 Working with targets

Compiler option
```
"$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>:-fopenmp>$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CUDA>:-Xcompiler=-fopenmp>"
```

Use a function that will fix a C++ only target by wrapping the flags if using a CUDA compiler
```
function(CUDA_CONVERT_FLAGS EXISTING_TARGET)
    get_property(old_flags TARGET ${EXISTING_TARGET} PROPERTY INTERFACE_COMPILE_OPTIONS)
    if(NOT "${old_flags}" STREQUAL "")
        string(REPLACE ";" "," CUDA_flags "${old_flags}")
        set_property(TARGET ${EXISTING_TARGET} PROPERTY INTERFACE_COMPILE_OPTIONS

       "$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CXX>:${old_flags}>$<$<BUILD_INTERFACE:$<COMPILE_LANGUAGE:CUDA>:-Xcompiler=${CU
            )
    endif()
endfunction()
```

2.4.3.6 Useful variables

- `CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES`: Place for built-in Thrust, etc
- `CMAKE_CUDA_COMPILER`: NVCC with location

```
set(CUDA_DIR "/usr/local/cuda-10.0")
set(CMAKE_CUDA_COMPILER ${CUDA_DIR}/bin/nvcc)
set(CMAKE_CUDA_FLAGS "-ccbin ${CC} -x cu -c -dc -O3  -Xcompiler "-O3 -pthread" -Wno-deprecated-gpu-targets
      -DVERSION=\"${GIT_VERSION}\"  --ptxas-options=-v")
set(CMAKE_CUDA_FLAGS_DEBUG ...)
set(CMAKE_CUDA_HOST_COMPILER ...)
set(CMAKE_CUDA_EXTENSIONS ...)
set(CMAKE_CUDA_STANDARD ...)
set(CMAKE_CUDA_RUNTIME_LIBRARY ...)
...
```

2.4.4 OpenMP

2.4.4.1 Enable OpenMP support

```
find_package(OpenMP)
if(OpenMP_CXX_FOUND)
    target_link_libraries(MyTarget PUBLIC OpenMP::OpenMP_CXX)
endif()
```

2.4.5 Boost

The Boost library is included in the find packages that CMake provides.

(Common) Settings related to boost

- `set(Boost_USE_STATIC_LIBS OFF)`
- `set(Boost_USE_MULTITHREADED ON)`
- `` `set(Boost_USE_STATIC_RUNTIME OFF) ``

E.g.: using the `Boost::filesystem` library

```
set(Boost_USE_STATIC_LIBS OFF)
set(Boost_USE_MULTITHREADED ON)
set(Boost_USE_STATIC_RUNTIME OFF)
find_package(Boost 1.50 REQUIRED COMPONENTS filesystem)
message(STATUS "Boost version: ${Boost_VERSION}")
# This is needed if your Boost version is newer than your CMake version
# or if you have an old version of CMake (<3.5)
if(NOT TARGET Boost::filesystem)
    add_library(Boost::filesystem IMPORTED INTERFACE)
    set_property(TARGET Boost::filesystem PROPERTY
        INTERFACE_INCLUDE_DIRECTORIES ${Boost_INCLUDE_DIR})
    set_property(TARGET Boost::filesystem PROPERTY
        INTERFACE_LINK_LIBRARIES ${Boost_LIBRARIES})
endif()
```

2.4.6 MPI

2.4.6.1 Enable MPI support

```
find_package(MPI REQUIRED)
message(STATUS "Run: ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} ${MPIEXEC_MAX_NUMPROCS} ${MPIEXEC_PREFLAGS}
       EXECUTABLE ${MPIEXEC_POSTFLAGS} ARGS")
target_link_libraries(MyTarget PUBLIC MPI::MPI_CXX)
```

2.5 Adding features

2.5.1 Set default build type

```
set(default_build_type "Release")
if(NOT CMAKE_BUILD_TYPE AND NOT CMAKE_CONFIGURATION_TYPES)
  message(STATUS "Setting build type to '${default_build_type}' as none was specified.")
  set(CMAKE_BUILD_TYPE "${default_build_type}" CACHE
      STRING "Choose the type of build." FORCE)
  # Set the possible values of build type for cmake-gui
  set_property(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS
    "Debug" "Release" "MinSizeRel" "RelWithDebInfo")
endif()
```

2.5.2 Meta compiler features

```
target_compile_features(myTarget PUBLIC cxx_std_11)
set_target_properties(myTarget PROPERTIES CXX_EXTENSIONS OFF)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
set_target_properties(myTarget PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED YES
    CXX_EXTENSIONS NO
)
```

2.5.3 Position independent code (-fPIC)

```
set(CMAKE_POSITION_INDEPENDENT_CODE ON)
# or target dependent
set_target_properties(lib1 PROPERTIES POSITION_INDEPENDENT_CODE ON)
```

2.5.4 Little libraries

```
find_library(MATH_LIBRARY m)
if(MATH_LIBRARY)
    target_link_libraries(MyTarget PUBLIC ${MATH_LIBRARY})
endif()
```

2.5.5 Modules

2.5.5.1 CMakeDependentOption

```
include(CMakeDependentOption)
cmake_dependent_option(BUILD_TESTS "Build your tests" ON "VAL1;VAL2" OFF)
```

which is equivalent to

```
if(VAL1 AND VAL2)
    set(BUILD_TESTS_DEFAULT ON)
else()
    set(BUILD_TESTS_DEFAULT OFF)
endif()
option(BUILD_TESTS "Build your tests" ${BUILD_TESTS_DEFAULT})
if(NOT BUILD_TESTS_DEFAULT)
    mark_as_advanced(BUILD_TESTS)
endif()
```

2.5.5.2 CMakePrintHelpers

```
cmake_print_properties
cmake_print_variables
```

2.5.5.3 CheckCXXCompilerFlag

Check whether flag is supported

```
include(CheckCXXCompilerFlag)
check_cxx_compiler_flag(-someflag OUTPUT_VARIABLE)
```

2.5.5.4 WriteCompilerDetectionHeader

Look for a list of features that some compilers support and write out a C++ header file that lets you know whether that feature is available

```
write_compiler_detection_header(
  FILE myoutput.h
  PREFIX My
  COMPILERS GNU Clang MSVC Intel
  FEATURES cxx_variadic_templates
)
```

2.5.5.5 try_compile / try_run

```
try_compile(
  RESULT_VAR
    bindir
  SOURCES
    source.cpp
)
```

2.6 Debugging

2.6.1 Printing variables

```
message(STATUS "MY_VARIABLE=${MY_VARIABLE}")
# or using module
include(CMakePrintHelpers)
cmake_print_variables(MY_VARIABLE)
cmake_print_properties(
    TARGETS my_target
    PROPERTIES POSITION_INDEPENDENT_CODE
)
```

2.6.2 Tracing a run

```
cmake -S . -B build --trace-source=CMakeLists.txt #--trace-expand
```

2.7 Including projects

2.7.1 Fetch

E.g.: download Catch2

```
FetchContent_Declare(
  catch
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG        v2.13.0
)
# CMake 3.14+
FetchContent_MakeAvailable(catch)
```

2.8 Testing

2.8.1 General

Enable testing and set a BUILD_TESTING option
```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME)
    include(CTest)
endif()
```

Add test folder
```
if(CMAKE_PROJECT_NAME STREQUAL PROJECT_NAME AND BUILD_TESTING)
    add_subdirectory(tests)
endif()
```

Register targets
```
add_test(NAME TestName COMMAND TargetName)
add_test(NAME TestName COMMAND $<TARGET_FILE:${TESTNAME}>)
```

2.8.2 Building as part of the test

```
add_test(
  NAME
    ExampleCMakeBuild
  COMMAND
    "${CMAKE_CTEST_COMMAND}"
              --build-and-test "${My_SOURCE_DIR}/examples/simple"
                               "${CMAKE_CURRENT_BINARY_DIR}/simple"
              --build-generator "${CMAKE_GENERATOR}"
              --test-command "${CMAKE_CTEST_COMMAND}"
)
```

2.8.3 Testing frameworks

2.8.3.1 GoogleTest

See Modern CMake: GoogleTest for reference.

Checkout GoogleTest as submodule
```
git submodule add --branch=release-1.8.0 ../../google/googletest.git extern/googletest
option(PACKAGE_TESTS "Build the tests" ON)
if(PACKAGE_TESTS)
    enable_testing()
    include(GoogleTest)
    add_subdirectory(tests)
endif()
```

2.8.3.2 Catch2

```
# Prepare "Catch" library for other executables
set(CATCH_INCLUDE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/extern/catch)
add_library(Catch2::Catch IMPORTED INTERFACE)
set_property(Catch2::Catch PROPERTY INTERFACE_INCLUDE_DIRECTORIES "${CATCH_INCLUDE_DIR}")
```

2.8.3.3 DocTest

DocTest is a replacement for *Catch2* that is supposed to compile much faster and be cleaner. Just replace *Catch2* with *DocTest*.

2.9 Exporting and Installing

Allow others to use your library, via

- *Bad way:* Find module
- *Add subproject:* `add_library(MyLib::MyLib ALIAS MyLib)`
- *Exporting:* Using ∗Config.cmake scripts

2.9.1 Installing

See GitBook: Installing Basic target install command (executed by e.g. `make install`)

```
install(TARGETS MyLib
        EXPORT MyLibTargets
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib
        RUNTIME DESTINATION bin
        INCLUDES DESTINATION include
        )
```

2.9.2 Exporting

See GitBook: Exporting

2.9.3 Packaging

See GitBook: Packaging

Chapter 3

Markdown cheatsheet

Short reference sheet for Markdown. Be aware that some things may not work properly in dependence of the used Markdown flavor.

3.1 Header 1

3.1.1 Header 2

3.1.1.1 Header 3

3.1.1.1.1 Header 4

Header 5

3.2 Emphasis

Emphasis, aka italics, with *asterisks* or *underscores*.

Strong emphasis, aka bold, with **asterisks** or **underscores**.

Combined emphasis with **asterisks and *underscores***.

Strikethrough uses two tildes. ~~Scratch this.~~

3.3 Lists

1. First ordered list item
2. Another item
    - Unordered sub-list.

1. Actual numbers don't matter, just that it's a number
    (a) Ordered sub-list
2. And another item.

    You can have properly indented paragraphs within list items. Notice the blank line above, and the leading spaces (at least one, but we'll use three here to also align the raw Markdown).

    To have a line break without a paragraph, you will need to use two trailing spaces. Note that this line is separate, but within the same paragraph. (This is contrary to the typical GFM line break behaviour, where trailing spaces are not required.)

- Unordered list can use asterisks
- Or minuses
- Or pluses

3.4 Links

`I'm an inline-style link`

`I'm an inline-style link with title`

`I'm a reference-style link`

`You can use numbers for reference-style link definitions`

Or leave it empty and use the `link text itself`.

URLs and URLs in angle brackets will automatically get turned into links. `http://www.example.com` or `http://www.example.com` and sometimes example.com (but not on Github, for example).

Some text to show that the reference links can follow later.

3.5 Images

Here's our logo (hover to see the title text):

Inline-style:

Reference-style:

3.6 Code and Syntax Highlighting

Inline `code` has `back-ticks around` it.

```
var s = "JavaScript syntax highlighting";
alert(s);
s = "Python syntax highlighting"
print(s)
No language indicated, so no syntax highlighting.
But let's throw in a <b>tag</b>.
```

3.7 Tables

Colons can be used to align columns.

Tables	Are	Cool
col 3 is	right-aligned	$1600
col 2 is	centered	$12
zebra stripes	are neat	$1

There must be at least 3 dashes separating each header cell. The outer pipes (|) are optional, and you don't need to make the raw Markdown line up prettily. You can also use inline Markdown.

Markdown	Less	Pretty
Still	`renders`	**nicely**
1	2	3

3.8 Blockquotes

> Blockquotes are very handy in email to emulate reply text. This line is part of the same quote.

Quote break.

> This is a very long line that will still be quoted properly when it wraps. Oh boy let's keep writing to make sure this is long enough to actually wrap for everyone. Oh, you can *put* **Markdown** into a blockquote.

3.9 Inline HTML

You can also use raw HTML in your Markdown, and it'll mostly work pretty well.

Definition list Is something people use sometimes.

Markdown in HTML Does *not* work **very** well. Use HTML *tags*.

3.10 Horizontal

Three or more...

Hyphens

Asterisks

Underscores

3.11 YouTube Videos

They can't be added directly but you can add an image with a link to the video like this:

Or, in pure Markdown, but losing the image sizing and border:

Referencing a bug by #bugID in your git commit links it to the slip. For example #1.

Chapter 4

Project structure

4.1 Folders

- **bin**: output executables go here (for the app, tests and spikes)
- **build**: containing all the object files (removed by clean)
- **doc**: documentation files
- **include**: all project header files, all necessary third-party header files (which are not in /usr/local/include)
- **lib**: any library that get compiled by the project, third party or any needed in development
- **spike**: smaller classes or files to test technologies or ideas
- **src**: the application and application's source files
- **test**: all test code files

4.2 Files

- **Makefile**: Makefile
- **README.md**: Readme file in markdown syntax

`CMake introduction: project structure`

- project
  - .gitignore
  - README.md
  - LICENCE.md
  - CMakeLists.txt
  - cmake
    - FindSomeLib.cmake
    - something_else.cmake
  - include
    - project
      - lib.hpp
  - src
    - CMakeLists.txt
    - lib.cpp
  - apps

    - CMakeLists.txt
    - app.cpp
- tests
    - CMakeLists.txt
    - testlib.cpp
- docs
    - CMakeLists.txt
- extern
    - googletest
- scripts
    - helper.py

Chapter 5

Unit-Tests

5.1 Integrated in CLion

5.1.1 Google Test

See Googletest - google Testing and Mocking Framework `Google test` on Github.

5.1.2 Catch

See `Catch Org` and `Catch2` for a modern, C++ native, header only test framework for unit-tests, TDD and BDD.

5.1.3 Boost.Test

See the `Boost.test` for the C++ Boost.Test library, providing both an easy to use and flexible set of interfaces for writing test programs, organizing tests into simple test cases and test suites, and controlling their runtime execution.

5.1.4 Doctest

`Doctest` is a new C++ testing framework but is by far the fastest both in compile times (by orders of magnitude) and runtime compared to other feature-rich alternatives. It brings the ability of compiled languages such as D / Rust / Nim to have tests written directly in the production code thanks to a fast, transparent and flexible test runner with a clean interface.

Chapter 6

Bug List

Member main ()

Bugs ...

Chapter 7

Todo List

Member main ()

add a

- add b
- add c

-

Chapter 8

Test List

Member main ()

Describing test case ...

Chapter 9

Namespace Index

9.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 10

Class Index

10.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 11

File Index

11.1 File List

Here is a list of all files with brief descriptions:

Chapter 12

Namespace Documentation

12.1 constants Namespace Reference

Variables

- constexpr double pi { 3.141519}
- constexpr double avogadro { 6.0221413e23 }

12.1.1 Variable Documentation

12.1.1.1 avogadro

```
constexpr double constants::avogadro { 6.0221413e23 }  [constexpr]
```

Definition at line 11 of file constants.h.

12.1.1.2 pi

```
constexpr double constants::pi { 3.141519}  [constexpr]
```

Definition at line 10 of file constants.h.

Chapter 13

Class Documentation

13.1 ConceptClass Class Reference

`#include "ConceptClass.h"`

Public Member Functions

- ConceptClass (int a, int b)

Public Attributes

- int member_a
- int member_b

13.1.1 Detailed Description

Definition at line 12 of file ConceptClass.h.

13.1.2 Constructor & Destructor Documentation

13.1.2.1 ConceptClass()

```
ConceptClass::ConceptClass (
            int a,
            int b )
```

Constructor
Detailed description for constructor.

Parameters

a	
b	

Definition at line 3 of file ConceptClass.cpp.

```
00003                                           {
00004     member_a = a;
00005     member_b = b;
00006 }
```

13.1.3 Member Data Documentation

13.1.3.1 member_a

```
int ConceptClass::member_a
```

Parameters

member	a

Definition at line 22 of file ConceptClass.h.

13.1.3.2 member_b

```
int ConceptClass::member_b
```

Parameters

member	b

Definition at line 24 of file ConceptClass.h.
The documentation for this class was generated from the following files:

- include/ConceptClass.h
- learningCpp/ConceptClass.cpp

Chapter 14

File Documentation

14.1 documents/CMakeIntroduction.md File Reference

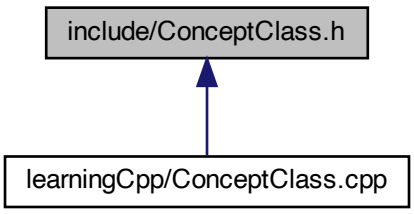
14.2 documents/Markdown.md File Reference

14.3 documents/structure.md File Reference

14.4 documents/Unit-Tests.md File Reference

14.5 include/ConceptClass.h File Reference

This graph shows which files directly or indirectly include this file:



Classes

- class ConceptClass

14.6 ConceptClass.h

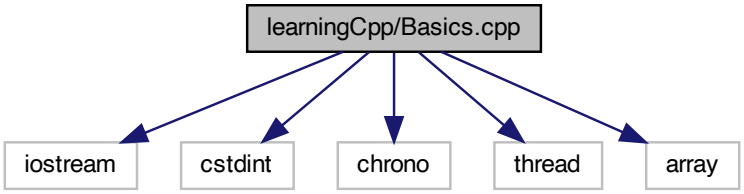
```
00001 #ifndef CPP_CONCEPTS_PROJECT_CONCEPTCLASS_H
00002 #define CPP_CONCEPTS_PROJECT_CONCEPTCLASS_H
00003
00012 class ConceptClass {
00013 public:
00019     ConceptClass(int a,int b);
00020
00022     int member_a;
00024     int member_b;
00025 };
00026
00027
```

```
00028 #endif //CPP_CONCEPTS_PROJECT_CONCEPTCLASS_H
```

14.7 learningCpp/Basics.cpp File Reference

```
#include <iostream>
#include <cstdint>
#include <chrono>
#include <thread>
#include <array>
```

Include dependency graph for Basics.cpp:



Functions

- int main ()

Variables

- int g_global_integer { 1 }
- static int g_x_1
- const int g_x_2 { 2 }

14.7.1 Function Documentation

14.7.1.1 main()

```
int main ( )
```

include order Initialization
Fundamental data types
escape sequences
Conditional operator
Namespaces
Static local variables
Typedefs and type aliases
Type conversion
Enumerations
Structs
Control flows
Arrays

Definition at line 31 of file Basics.cpp.

```
00031           {
00032
00034     // copy initialization
00035     int a = 1;
```

```
00036     // direct initialization
00037     int b(1);
00038     // list (uniform/brace) initialization
00039     //direct
00040     int c_1{1};
00041     //copy
00042     int c_2 = {1};
00046     // floating point
00047     float float_a = 3.14159; // at least 4 bytes
00048     double float_b = 3.14159; // at least 8 bytes
00049     long double float_c = 3.14159; // at least 8 bytes
00050     // Inf represents Infinity
00051     // NaN represents Not a Number
00052
00053     // integral characters
00054     char char_a = 'c'; // always 1 byte
00055     wchar_t char_b = 'c'; // at least 1 byte
00056     //char8_t char_c = 'c'; // C++20
00057     //char16_t char_d = 'c'; // C++11 // at least 2 bytes
00058     //char32_t char_e = 'c'; // C++11 // at least 4 bytes
00059
00060     // 0b12 --> binary
00061     // 012 --> octal
00062     // 0x12 --> hexadecimal
00063     // use std::dec , std::oct , std::hex
00064
00065     // Integers
00066     short int_a = 1; // at least 2 bytes
00067     int int_b = 1; // at least 2 bytes
00068     long int_c = 1; // at least 4 bytes
00069     //long long int_d = 1; // C++11
00070
00071     // Boolean
00072     bool bool_a = true; // or false
00073
00074     // Null pointer
00075     //std::nullptr_t null_pointer = nullptr;
00076
00077     // void
00078
00079     // using cstdint
00080     //std::int8_t
00081     //std::uint8_t
00082     //std::int16_t
00083     //std::uint16_t
00084     //std::int32_t
00085     //std::uint32_t
00086     //std::int64_t
00087     //std::uint64_t
00088
00089     // there is also the std::int_fast#_t providing the fastest signed integer with at least # bits
00090     // there is also the std::int_least#_t providing the smallest signed integer with at least # bits
00091
00092     std::cout « "bool:\t\t" « sizeof(bool) « " bytes\n";
00093     std::cout « "char:\t\t" « sizeof(char) « " bytes\n";
00094     std::cout « "wchar_t:\t" « sizeof(wchar_t) « " bytes\n";
00095     std::cout « "char16_t:\t" « sizeof(char16_t) « " bytes\n"; // C++11 only
00096     std::cout « "char32_t:\t" « sizeof(char32_t) « " bytes\n"; // C++11 only
00097     std::cout « "short:\t\t" « sizeof(short) « " bytes\n";
00098     std::cout « "int:\t\t" « sizeof(int) « " bytes\n";
00099     std::cout « "long:\t\t" « sizeof(long) « " bytes\n";
00100     std::cout « "long long:\t" « sizeof(long long) « " bytes\n"; // C++11 only
00101     std::cout « "float:\t\t" « sizeof(float) « " bytes\n";
00102     std::cout « "double:\t\t" « sizeof(double) « " bytes\n";
00103     std::cout « "long double:\t" « sizeof(long double) « " bytes\n";
00104
00105     // use const
00106     //const int const_int = 1;
00107     // for variables that should not be modifiable after initialization
00108     // and whose initializer is NOT known at compile-time
00109
00110     // use constexpr
00111     //constexpr int constexpr_int = 1;
00112     // for variables that should not be modifiable after initialization
00113     // and whose initializer is known at compile-time
00117     for (int i = 0; i < 5; i++) {
00118         std::this_thread::sleep_for(std::chrono::milliseconds(250));
00119         std::cout « "\a"; // makes an alert
00120     }
00121     std::cout « "Backspace \b" « std::endl;
00122     std::cout « "Formfeed \f" « std::endl;
00123     std::cout « "Newline \n" « std::endl;
00124     std::cout « "Carriage return \r" « std::endl;
00125     std::cout « "Horizontal \t tab" « std::endl;
00126     std::cout « "Vertical tab \v" « std::endl;
00127     std::cout « "Single quote \' or double quote \"" « std::endl;
00128     std::cout « "Octal number \12" « std::endl;
```

```
00129     std::cout « "Hex number \x14" « std::endl;
00133     int x_1 = 2;
00134     int x_2 = 3;
00135     int max_x = (x_1 > x_2) ? x_1 : x_2;
00139     // define a namespace
00140     //namespace namespace_1 {
00141     //    //nested namespace
00142     //    namespace namespace_1_nested {
00143
00144     //    }
00145     //}
00146     // accessible using "::"
00147
00148     // namespace alias
00149     // namespace nested_namespace = namespace_1::namespace_1_nested;
00154     // static local variables are not destroyed when out of scope (in contrast to automatic)
00155     static int var_1 { 1 };
00156     // AVOID using static variables unless the variable never needs to be reset
00161     typedef double distance_t; // define distance_t as an alias for type double
00162     //which is equivalent to: using distance_t = double;
00163     // The following two statements are equivalent:
00164     // double howFar; //equivalent to
00165     distance_t howFar;
00171     // IMPLICIT type conversion (coercion)
00172     float f_int { 3 }; // initializing floating point variable with int 3
00173
00174     // EXPLICIT type conversion
00175     // static_cast
00176     int i1 { 10 };
00177     int i2 { 4 };
00178     // convert an int to a float so we get floating point division rather than integer division
00179     float f { static_cast<float>(i1) / i2 };
00180
00184     enum Color
00185     {
00186         color_black, // assigned 0
00187         color_red, // assigned 1
00188         color_blue, // assigned 2
00189         color_green, // assigned 3
00190         color_white, // assigned 4
00191         color_cyan, // assigned 5
00192         color_yellow, // assigned 6
00193         color_magenta // assigned 7
00194     };
00195     Color paint{ color_white };
00196     std::cout « paint;
00197
00198     // enum classes (scoped enumerations)
00199     enum class Fruit
00200     {
00201         banana, // banana is inside the scope of Fruit
00202         apple
00203     };
00204     Fruit fruit{ Fruit::banana }; // note: banana is not directly accessible any more, we have to use
    Fruit::banana
00208     struct Employee
00209     {
00210         short id;
00211         int age;
00212         double wage;
00213     };
00214
00215     Employee joe{ 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
00216     Employee frank{ 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default
    initialization)
00217
00218     //Employee joe; // create an Employee struct for Joe
00219     //joe.id = 14; // assign a value to member id within struct joe
00220     //joe.age = 32; // assign a value to member age within struct joe
00221     //joe.wage = 24.15; // assign a value to member wage within struct joe
00222
00223     //Employee frank; // create an Employee struct for Frank
00224     //frank.id = 15; // assign a value to member id within struct frank
00225     //frank.age = 28; // assign a value to member age within struct frank
00226     //frank.wage = 18.27; // assign a value to member wage within struct frank
00227
00228     // nested structs
00229     struct Company
00230     {
00231         Employee CEO; // Employee is a struct within the Company struct
00232         int numberOfEmployees;
00233     };
00234     Company myCompany{{ 1, 42, 60000.0 }, 5 };
00238     // halt (using <cstdlib>)
00239     //std::exit(0); // terminate and return 0 to operating system
00240     // ATTENTION: be aware of leaking resources
00241
```

```
00242     // Conditional branches
00243     if (true) {
00244
00245     } else if (false) {
00246
00247     } else {
00248
00249     }
00250     // init statements
00251     //    if (std::string fullName{ firstName + ' ' + lastName }; fullName.length() > 20)
00252     //    {
00253     //        std::cout « '"' « fullName « "\"is too long!\n";
00254     //    }
00255     //    else
00256     //    {
00257     //        std::cout « "Your name is " « fullName « '\n';
00258     //    }
00259
00260     // Switch statements
00261     Color color {color_black};
00262     switch (color)
00263     {
00264         case Color::color_black:
00265             std::cout « "Black";
00266             break;
00267         case Color::color_white:
00268             std::cout « "White";
00269             break;
00270         case Color::color_red:
00271             std::cout « "Red";
00272             break;
00273             //[[fallthrough]];
00274         case Color::color_green:
00275             std::cout « "Green";
00276             break;
00277         case Color::color_blue:
00278             std::cout « "Blue";
00279             break;
00280         default:
00281             std::cout « "Unknown";
00282             break;
00283     }
00284     //[[fallthrough]] attribute can be added to indicate that the fall-through is intentional.
00285
00286     // Goto statements
00287     //tryAgain:
00288     //    goto tryAgain;
00289
00290     // While statements
00291     int while_counter{ 5 };
00292     while (while_counter < 10) {
00293         std::cout « "while_counter: " « while_counter « std::endl;
00294         ++while_counter;
00295     }
00296
00297     // Do wile statements
00298     do {
00299         std::cout « "while_counter: " « while_counter « std::endl;
00300         ++while_counter;
00301     }
00302     while (while_counter < 15);
00303
00304     // For statements
00305     for (int count{ 0 }; count < 10; ++count)
00306         std::cout « count « ' ';
00307     int iii{};
00308     int jjj{};
00309     for (iii = 0, jjj = 9; iii < 10; ++iii, --jjj)
00310         std::cout « iii « ' ' « jjj « '\n';
00311     // return statement terminates the entire function the loop is within
00312     // break terminates the loop
00313     // continue jumps to the end of the loop body for the current iteration
00317     //int prime[5]{}; // hold the first 5 prime numbers
00318     //prime[0] = 2; // The first element has index 0
00319     //prime[1] = 3;
00320     //prime[2] = 5;
00321     //prime[3] = 7;
00322     //prime[4] = 11; // The last element has index 4 (array length-1)
00323     int prime[5]{ 2, 3, 5, 7, 11 }; // use initializer list to initialize the fixed array
00324     //int prime[]{ 2, 3, 5, 7, 11 }; // works as well
00325     //std::cout « "The array has: " « std::size(prime) « " elements\n"; // C++17
00326     //sizeof() gives the array length multiplied by element size
00327
00328     // Multidimensional arrays
00329     int num_rows{3};
00330     int num_cols{5};
00331     int multi_dim_array[3][5] // cannot use num_rows or num_cols --> see dynamic memory allocation
```

```
00332             {
00333                     { 1, 2, 3, 4, 5 }, // row 0
00334                     { 6, 7, 8, 9, 10 }, // row 1
00335                     { 11, 12, 13, 14, 15 } // row 2
00336             };
00337     for (int row{ 0 }; row < num_rows; ++row) // step through the rows in the array
00338     {
00339         for (int col{ 0 }; col < num_cols; ++col) // step through each element in the row
00340         {
00341             std::cout « multi_dim_array[row][col];
00342         }
00343     }
00344
00345     // foreach loop
00346     for (auto &element: prime)
00347     {
00348         std::cout «  element « std::endl;
00349     }
00352     return 0; // 0, EXIT_SUCCESS, EXIT_FAILURE
00353 }
```

14.7.2 Variable Documentation

14.7.2.1 g_global_integer

```
int g_global_integer { 1 }
```

Global variables
Definition at line 13 of file Basics.cpp.

14.7.2.2 g_x_1

```
int g_x_1  [static]
```

Definition at line 18 of file Basics.cpp.

14.7.2.3 g_x_2

```
const int g_x_2 { 2 }  [extern]
```

14.8 Basics.cpp

```
00001 //
00002 // Created by Michael Staneker on 01.12.20.
00003 //
00004
00005 #include <iostream>
00006 #include <cstdint>
00007 #include <chrono>
00008 #include <thread>
00009 #include <array>
00010
00012 // global variables have file scope
00013 int g_global_integer { 1 };
00014 // AVOID using non-constant global variables!
00015
00016 // internal linkage --> limits the use of an identifier to a single file
00017 // non-constant globals have external linkage by default
00018 static int g_x_1; // adding static makes them internal linkage
00019 // const & constexpr globals have internal linkage by default
00020
00021 // external linkage --> "truly global"
00022 // functions have external linkage by default!
00023 extern const int g_x_2 { 2 }; // making const external
00027 // user-defined headers (alphabetically)
00028 // third-party library headers (alphabetically)
00029 // standard library header (alphabetically)
00030
00031 int main() {
00032
00034     // copy initialization
00035     int a = 1;
00036     // direct initialization
```

```
00037     int b(1);
00038     // list (uniform/brace) initialization
00039     //direct
00040     int c_1{1};
00041     //copy
00042     int c_2 = {1};
00046     // floating point
00047     float float_a = 3.14159; // at least 4 bytes
00048     double float_b = 3.14159; // at least 8 bytes
00049     long double float_c = 3.14159; // at least 8 bytes
00050     // Inf represents Infinity
00051     // NaN represents Not a Number
00052
00053     // integral characters
00054     char char_a = 'c'; // always 1 byte
00055     wchar_t char_b = 'c'; // at least 1 byte
00056     //char8_t char_c = 'c'; // C++20
00057     //char16_t char_d = 'c'; // C++11 // at least 2 bytes
00058     //char32_t char_e = 'c'; // C++11 // at least 4 bytes
00059
00060     // 0b12 --> binary
00061     // 012 --> octal
00062     // 0x12 --> hexadecimal
00063     // use std::dec , std::oct , std::hex
00064
00065     // Integers
00066     short int_a = 1; // at least 2 bytes
00067     int int_b = 1; // at least 2 bytes
00068     long int_c = 1; // at least 4 bytes
00069     //long long int_d = 1; // C++11
00070
00071     // Boolean
00072     bool bool_a = true; // or false
00073
00074     // Null pointer
00075     //std::nullptr_t null_pointer = nullptr;
00076
00077     // void
00078
00079     // using cstdint
00080     //std::int8_t
00081     //std::uint8_t
00082     //std::int16_t
00083     //std::uint16_t
00084     //std::int32_t
00085     //std::uint32_t
00086     //std::int64_t
00087     //std::uint64_t
00088
00089     // there is also the std::int_fast#_t providing the fastest signed integer with at least # bits
00090     // there is also the std::int_least#_t providing the smallest signed integer with at least # bits
00091
00092     std::cout « "bool:\t\t" « sizeof(bool) « " bytes\n";
00093     std::cout « "char:\t\t" « sizeof(char) « " bytes\n";
00094     std::cout « "wchar_t:\t" « sizeof(wchar_t) « " bytes\n";
00095     std::cout « "char16_t:\t" « sizeof(char16_t) « " bytes\n"; // C++11 only
00096     std::cout « "char32_t:\t" « sizeof(char32_t) « " bytes\n"; // C++11 only
00097     std::cout « "short:\t\t" « sizeof(short) « " bytes\n";
00098     std::cout « "int:\t\t" « sizeof(int) « " bytes\n";
00099     std::cout « "long:\t\t" « sizeof(long) « " bytes\n";
00100     std::cout « "long long:\t" « sizeof(long long) « " bytes\n"; // C++11 only
00101     std::cout « "float:\t\t" « sizeof(float) « " bytes\n";
00102     std::cout « "double:\t\t" « sizeof(double) « " bytes\n";
00103     std::cout « "long double:\t" « sizeof(long double) « " bytes\n";
00104
00105     // use const
00106     //const int const_int = 1;
00107     // for variables that should not be modifiable after initialization
00108     // and whose initializer is NOT known at compile-time
00109
00110     // use constexpr
00111     //constexpr int constexpr_int = 1;
00112     // for variables that should not be modifiable after initialization
00113     // and whose initializer is known at compile-time
00117     for (int i = 0; i < 5; i++) {
00118         std::this_thread::sleep_for(std::chrono::milliseconds(250));
00119         std::cout « "\a"; // makes an alert
00120     }
00121     std::cout « "Backspace \b" « std::endl;
00122     std::cout « "Formfeed \f" « std::endl;
00123     std::cout « "Newline \n" « std::endl;
00124     std::cout « "Carriage return \r" « std::endl;
00125     std::cout « "Horizontal \t tab" « std::endl;
00126     std::cout « "Vertical tab \v" « std::endl;
00127     std::cout « "Single quote \' or double quote \"" « std::endl;
00128     std::cout « "Octal number \12" « std::endl;
00129     std::cout « "Hex number \x14" « std::endl;
```

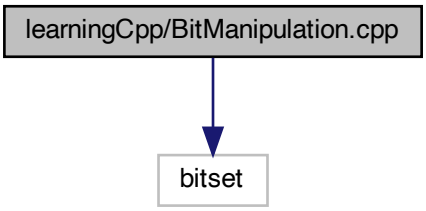
```
00133     int x_1 = 2;
00134     int x_2 = 3;
00135     int max_x = (x_1 > x_2) ? x_1 : x_2;
00139     // define a namespace
00140     //namespace namespace_1 {
00141     //    //nested namespace
00142     //    namespace namespace_1_nested {
00143
00144     //    }
00145     //}
00146     // accessible using "::"
00147
00148     // namespace alias
00149     // namespace nested_namespace = namespace_1::namespace_1_nested;
00154     // static local variables are not destroyed when out of scope (in contrast to automatic)
00155     static int var_1 { 1 };
00156     // AVOID using static variables unless the variable never needs to be reset
00161     typedef double distance_t; // define distance_t as an alias for type double
00162     //which is equivalent to: using distance_t = double;
00163     // The following two statements are equivalent:
00164     // double howFar; //equivalent to
00165     distance_t howFar;
00171     // IMPLICIT type conversion (coercion)
00172     float f_int { 3 }; // initializing floating point variable with int 3
00173
00174     // EXPLICIT type conversion
00175     // static_cast
00176     int i1 { 10 };
00177     int i2 { 4 };
00178     // convert an int to a float so we get floating point division rather than integer division
00179     float f { static_cast<float>(i1) / i2 };
00180
00184     enum Color
00185     {
00186         color_black, // assigned 0
00187         color_red, // assigned 1
00188         color_blue, // assigned 2
00189         color_green, // assigned 3
00190         color_white, // assigned 4
00191         color_cyan, // assigned 5
00192         color_yellow, // assigned 6
00193         color_magenta // assigned 7
00194     };
00195     Color paint{ color_white };
00196     std::cout « paint;
00197
00198     // enum classes (scoped enumerations)
00199     enum class Fruit
00200     {
00201         banana, // banana is inside the scope of Fruit
00202         apple
00203     };
00204     Fruit fruit{ Fruit::banana }; // note: banana is not directly accessible any more, we have to use
      Fruit::banana
00208     struct Employee
00209     {
00210         short id;
00211         int age;
00212         double wage;
00213     };
00214
00215     Employee joe{ 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
00216     Employee frank{ 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default
      initialization)
00217
00218     //Employee joe; // create an Employee struct for Joe
00219     //joe.id = 14; // assign a value to member id within struct joe
00220     //joe.age = 32; // assign a value to member age within struct joe
00221     //joe.wage = 24.15; // assign a value to member wage within struct joe
00222
00223     //Employee frank; // create an Employee struct for Frank
00224     //frank.id = 15; // assign a value to member id within struct frank
00225     //frank.age = 28; // assign a value to member age within struct frank
00226     //frank.wage = 18.27; // assign a value to member wage within struct frank
00227
00228     // nested structs
00229     struct Company
00230     {
00231         Employee CEO; // Employee is a struct within the Company struct
00232         int numberOfEmployees;
00233     };
00234     Company myCompany{{ 1, 42, 60000.0 }, 5 };
00238     // halt (using <cstdlib>)
00239     //std::exit(0); // terminate and return 0 to operating system
00240     // ATTENTION: be aware of leaking resources
00241
00242     // Conditional branches
```

```
00243     if (true) {
00244
00245     } else if (false) {
00246
00247     } else {
00248
00249     }
00250     // init statements
00251     //    if (std::string fullName{ firstName + ' ' + lastName }; fullName.length() > 20)
00252     //    {
00253     //        std::cout « '"' « fullName « "\"is too long!\n";
00254     //    }
00255     //    else
00256     //    {
00257     //        std::cout « "Your name is " « fullName « '\n';
00258     //    }
00259
00260     // Switch statements
00261     Color color {color_black};
00262     switch (color)
00263     {
00264         case Color::color_black:
00265             std::cout « "Black";
00266             break;
00267         case Color::color_white:
00268             std::cout « "White";
00269             break;
00270         case Color::color_red:
00271             std::cout « "Red";
00272             break;
00273             //[[fallthrough]];
00274         case Color::color_green:
00275             std::cout « "Green";
00276             break;
00277         case Color::color_blue:
00278             std::cout « "Blue";
00279             break;
00280         default:
00281             std::cout « "Unknown";
00282             break;
00283     }
00284     //[[fallthrough]] attribute can be added to indicate that the fall-through is intentional.
00285
00286     // Goto statements
00287     //tryAgain:
00288     //    goto tryAgain;
00289
00290     // While statements
00291     int while_counter{ 5 };
00292     while (while_counter < 10) {
00293         std::cout « "while_counter: " « while_counter « std::endl;
00294         ++while_counter;
00295     }
00296
00297     // Do wile statements
00298     do {
00299         std::cout « "while_counter: " « while_counter « std::endl;
00300         ++while_counter;
00301     }
00302     while (while_counter < 15);
00303
00304     // For statements
00305     for (int count{ 0 }; count < 10; ++count)
00306         std::cout « count « ' ';
00307     int iii{};
00308     int jjj{};
00309     for (iii = 0, jjj = 9; iii < 10; ++iii, --jjj)
00310         std::cout « iii « ' ' « jjj « '\n';
00311     // return statement terminates the entire function the loop is within
00312     // break terminates the loop
00313     // continue jumps to the end of the loop body for the current iteration
00317     //int prime[5]{}; // hold the first 5 prime numbers
00318     //prime[0] = 2; // The first element has index 0
00319     //prime[1] = 3;
00320     //prime[2] = 5;
00321     //prime[3] = 7;
00322     //prime[4] = 11; // The last element has index 4 (array length-1)
00323     int prime[5]{ 2, 3, 5, 7, 11 }; // use initializer list to initialize the fixed array
00324     //int prime[]{ 2, 3, 5, 7, 11 }; // works as well
00325     //std::cout « "The array has: " « std::size(prime) « " elements\n"; // C++17
00326     //sizeof() gives the array length multiplied by element size
00327
00328     // Multidimensional arrays
00329     int num_rows{3};
00330     int num_cols{5};
00331     int multi_dim_array[3][5] // cannot use num_rows or num_cols --> see dynamic memory allocation
00332             {
```

```
00333                     { 1, 2, 3, 4, 5 }, // row 0
00334                     { 6, 7, 8, 9, 10 }, // row 1
00335                     { 11, 12, 13, 14, 15 } // row 2
00336             };
00337     for (int row{ 0 }; row < num_rows; ++row) // step through the rows in the array
00338     {
00339         for (int col{ 0 }; col < num_cols; ++col) // step through each element in the row
00340         {
00341             std::cout « multi_dim_array[row][col];
00342         }
00343     }
00344
00345     // foreach loop
00346     for (auto &element: prime)
00347     {
00348         std::cout «  element « std::endl;
00349     }
00352     return 0; // 0, EXIT_SUCCESS, EXIT_FAILURE
00353 }
```

14.9 learningCpp/BitManipulation.cpp File Reference

`#include <bitset>`
Include dependency graph for BitManipulation.cpp:



Functions

- int main ()

14.9.1 Function Documentation

14.9.1.1 main()

`int main ( )`
Bitwise operators
Bit masks
Definition at line 7 of file BitManipulation.cpp.

```
00007             {
00008
00009     std::bitset<8> bits{ 0b0000'0101 }; // we need 8 bits, start with bit pattern 0000 0101
00010     bits.set(3); // set bit position 3 to 1 (now we have 0000 1101)
00011     bits.flip(4); // flip bit 4 (now we have 0001 1101)
00012     bits.reset(4); // set bit 4 back to 0 (now we have 0000 1101)
00013
00014     std::cout « "All the bits: " « bits « '\n';
00015     std::cout « "Bit 3 has value: " « bits.test(3) « '\n';
00016     std::cout « "Bit 4 has value: " « bits.test(4) « '\n';
00017
00019     // x « y // left shift
00020     // x » y // right shift
00021     // ~x // bitwise NOT
00022     // x & y // bitwise AND
```

```
00023     // x | y // bitwise OR
00024     // x ^ y // bitwise XOR
00025     // x «= < // left shift assignment
00026     // x »= y // right shift assignment
00027     // x |= y // bitwise OR assignment
00028     // x &= y // bitwise AND assignment
00029     // x ^= y // bitwise XOR assignment
00033     // since C++14
00034     constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
00035     constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
00036     constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
00037     constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
00038     constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
00039     constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
00040     constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
00041     constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7
00042     // C++11 or earlier
00043     //    constexpr std::uint_fast8_t mask0{ 0x1 }; // hex for 0000 0001
00044     //    constexpr std::uint_fast8_t mask1{ 0x2 }; // hex for 0000 0010
00045     //    constexpr std::uint_fast8_t mask2{ 0x4 }; // hex for 0000 0100
00046     //    constexpr std::uint_fast8_t mask3{ 0x8 }; // hex for 0000 1000
00047     //    constexpr std::uint_fast8_t mask4{ 0x10 }; // hex for 0001 0000
00048     //    constexpr std::uint_fast8_t mask5{ 0x20 }; // hex for 0010 0000
00049     //    constexpr std::uint_fast8_t mask6{ 0x40 }; // hex for 0100 0000
00050     //    constexpr std::uint_fast8_t mask7{ 0x80 }; // hex for 1000 0000
00051     //    // or
00052     //    constexpr std::uint_fast8_t mask0{ 1 « 0 }; // 0000 0001
00053     //    constexpr std::uint_fast8_t mask1{ 1 « 1 }; // 0000 0010
00054     //    constexpr std::uint_fast8_t mask2{ 1 « 2 }; // 0000 0100
00055     //    constexpr std::uint_fast8_t mask3{ 1 « 3 }; // 0000 1000
00056     //    constexpr std::uint_fast8_t mask4{ 1 « 4 }; // 0001 0000
00057     //    constexpr std::uint_fast8_t mask5{ 1 « 5 }; // 0010 0000
00058     //    constexpr std::uint_fast8_t mask6{ 1 « 6 }; // 0100 0000
00059     //    constexpr std::uint_fast8_t mask7{ 1 « 7 }; // 1000 0000
00062     return 0;
00063 }
```

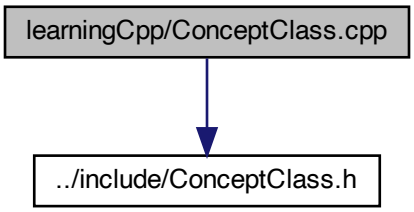
14.10 BitManipulation.cpp

```
00001 //
00002 // Created by Michael Staneker on 01.12.20.
00003 //
00004
00005 #include <bitset>
00006
00007 int main() {
00008
00009     std::bitset<8> bits{ 0b0000'0101 }; // we need 8 bits, start with bit pattern 0000 0101
00010     bits.set(3); // set bit position 3 to 1 (now we have 0000 1101)
00011     bits.flip(4); // flip bit 4 (now we have 0001 1101)
00012     bits.reset(4); // set bit 4 back to 0 (now we have 0000 1101)
00013
00014     std::cout « "All the bits: " « bits « '\n';
00015     std::cout « "Bit 3 has value: " « bits.test(3) « '\n';
00016     std::cout « "Bit 4 has value: " « bits.test(4) « '\n';
00017
00019     // x « y // left shift
00020     // x » y // right shift
00021     // ~x // bitwise NOT
00022     // x & y // bitwise AND
00023     // x | y // bitwise OR
00024     // x ^ y // bitwise XOR
00025     // x «= < // left shift assignment
00026     // x »= y // right shift assignment
00027     // x |= y // bitwise OR assignment
00028     // x &= y // bitwise AND assignment
00029     // x ^= y // bitwise XOR assignment
00033     // since C++14
00034     constexpr std::uint_fast8_t mask0{ 0b0000'0001 }; // represents bit 0
00035     constexpr std::uint_fast8_t mask1{ 0b0000'0010 }; // represents bit 1
00036     constexpr std::uint_fast8_t mask2{ 0b0000'0100 }; // represents bit 2
00037     constexpr std::uint_fast8_t mask3{ 0b0000'1000 }; // represents bit 3
00038     constexpr std::uint_fast8_t mask4{ 0b0001'0000 }; // represents bit 4
00039     constexpr std::uint_fast8_t mask5{ 0b0010'0000 }; // represents bit 5
00040     constexpr std::uint_fast8_t mask6{ 0b0100'0000 }; // represents bit 6
00041     constexpr std::uint_fast8_t mask7{ 0b1000'0000 }; // represents bit 7
00042     // C++11 or earlier
00043     //    constexpr std::uint_fast8_t mask0{ 0x1 }; // hex for 0000 0001
00044     //    constexpr std::uint_fast8_t mask1{ 0x2 }; // hex for 0000 0010
00045     //    constexpr std::uint_fast8_t mask2{ 0x4 }; // hex for 0000 0100
00046     //    constexpr std::uint_fast8_t mask3{ 0x8 }; // hex for 0000 1000
00047     //    constexpr std::uint_fast8_t mask4{ 0x10 }; // hex for 0001 0000
00048     //    constexpr std::uint_fast8_t mask5{ 0x20 }; // hex for 0010 0000
00049     //    constexpr std::uint_fast8_t mask6{ 0x40 }; // hex for 0100 0000
00050     //    constexpr std::uint_fast8_t mask7{ 0x80 }; // hex for 1000 0000
```

```
00051     //    // or
00052     //    constexpr std::uint_fast8_t mask0{ 1 << 0 }; // 0000 0001
00053     //    constexpr std::uint_fast8_t mask1{ 1 << 1 }; // 0000 0010
00054     //    constexpr std::uint_fast8_t mask2{ 1 << 2 }; // 0000 0100
00055     //    constexpr std::uint_fast8_t mask3{ 1 << 3 }; // 0000 1000
00056     //    constexpr std::uint_fast8_t mask4{ 1 << 4 }; // 0001 0000
00057     //    constexpr std::uint_fast8_t mask5{ 1 << 5 }; // 0010 0000
00058     //    constexpr std::uint_fast8_t mask6{ 1 << 6 }; // 0100 0000
00059     //    constexpr std::uint_fast8_t mask7{ 1 << 7 }; // 1000 0000
00062     return 0;
00063 }
00064
```

14.11 learningCpp/ConceptClass.cpp File Reference

`#include "../include/ConceptClass.h"`
Include dependency graph for ConceptClass.cpp:



14.12 ConceptClass.cpp

```
00001 #include "../include/ConceptClass.h"
00002
00003 ConceptClass::ConceptClass(int a, int b) {
00004     member_a = a;
00005     member_b = b;
00006 }
```

14.13 learningCpp/constants.h File Reference

Namespaces

- constants

Variables

- constexpr double constants::pi { 3.141519}
- constexpr double constants::avogadro { 6.0221413e23 }

14.14 constants.h

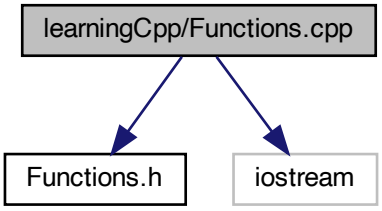
```
00001 //
00002 // Created by Michael Staneker on 01.12.20.
00003 //
00004
00005 #ifndef CPP_TEMPLATE_PROJECT_CONSTANTS_H
00006 #define CPP_TEMPLATE_PROJECT_CONSTANTS_H
00007
00008 namespace constants {
00009
```

```
00010     constexpr double pi { 3.141519};
00011     constexpr double avogadro { 6.0221413e23 };
00012
00013     //extern const double pi { 3.141519};
00014     //extern const double avogadro { 6.0221413e23 };
00015
00016     // C++17 or newer
00017     //inline constexpr double pi { 3.14159 }; // inline constexpr is C++17 or newer only
00018     //inline constexpr double avogadro { 6.0221413e23 };
00019
00020     //#include "constants.h"
00021     //
00022     //double circumfence { 2.0 * radius * constants::pi}
00023 }
00024
00025 #endif //CPP_TEMPLATE_PROJECT_CONSTANTS_H
```

14.15 learningCpp/Functions.cpp File Reference

```
#include "Functions.h"
#include <iostream>
```

Include dependency graph for Functions.cpp:



Functions

- void pass_by_value (int x)
- int main ()

14.15.1 Function Documentation

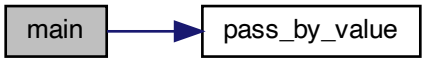
14.15.1.1 main()

```
int main ( )
```

Definition at line 14 of file Functions.cpp.

```
00014           {
00015
00016     int x = 5;
00017     pass_by_value(x);
00018
00019     return 0;
00020 }
```

Here is the call graph for this function:



14.15.1.2 pass_by_value()

```
void pass_by_value (
             int  )
```

14.15.2 Function parameters and arguments

14.15.2.1 Pass by value

By default, non-pointer arguments in C++ are passed by value. When an argument is **passed by value**, the argument's value is copied into the value of the corresponding function parameter. Therefore the original argument can not be modified by the function!

14.15.2.1.1 Pros

- Arguments can be anything
- Arguments are never

14.15.2.1.2 Cons

- Copying classes and structs can incur a significant performance penalty

14.15.2.1.3 When to use

- When passing fundamental data type and enumerators, and the function does not need to change the argument.

14.15.2.1.4 When not to use

- When passing structs or classes (including std::array, std::vector, and std::string) Function passing argument by value

Definition at line 9 of file Functions.cpp.

```
00009                          {
00010     std::cout « "func: pass_by_value(int x)" « std::endl;
00011     std::cout « "x = " « x « std::endl;
00012 }
```

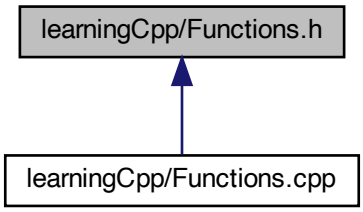
14.16 Functions.cpp

```
00001 //
00002 // Created by Michael Staneker on 08.12.20.
00003 //
00004
00005 #include "Functions.h"
00006
00007 #include <iostream>
00008
```

```
00009 void pass_by_value(int x) {
00010     std::cout « "func: pass_by_value(int x)" « std::endl;
00011     std::cout « "x = " « x « std::endl;
00012 }
00013
00014 int main() {
00015
00016     int x = 5;
00017     pass_by_value(x);
00018
00019     return 0;
00020 }
```

14.17 learningCpp/Functions.h File Reference

This graph shows which files directly or indirectly include this file:



Functions

- void pass_by_value (int)

14.17.1 Function Documentation

14.17.1.1 pass_by_value()

```
void pass_by_value (
            int  )
```

14.17.2 Function parameters and arguments

14.17.2.1 Pass by value

By default, non-pointer arguments in C++ are passed by value. When an argument is **passed by value**, the argument's value is copied into the value of the corresponding function parameter. Therefore the original argument can not be modified by the function!

14.17.2.1.1 Pros

- Arguments can be anything
- Arguments are never

14.17.2.1.2 Cons

- Copying classes and structs can incur a significant performance penalty

14.17.2.1.3 When to use

- When passing fundamental data type and enumerators, and the function does not need to change the argument.

14.17.2.1.4 When not to use

- When passing structs or classes (including std::array, std::vector, and std::string) Function passing argument by value

Definition at line 9 of file Functions.cpp.

```
00009                                {
00010     std::cout « "func: pass_by_value(int x)" « std::endl;
00011     std::cout « "x = " « x « std::endl;
00012 }
```

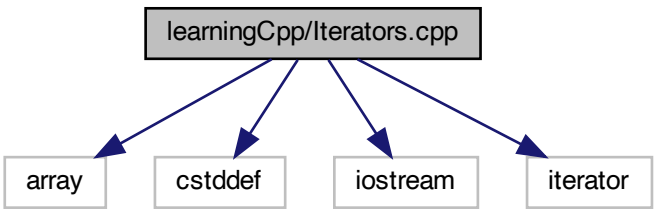
14.18 Functions.h

```
00001 //
00002 // Created by Michael Staneker on 08.12.20.
00003 //
00004
00005 #ifndef CPP_TEMPLATE_PROJECT_FUNCTIONS_H
00006 #define CPP_TEMPLATE_PROJECT_FUNCTIONS_H
00007
00040 void pass_by_value(int);
00041
00042 #endif //CPP_TEMPLATE_PROJECT_FUNCTIONS_H
```

14.19 learningCpp/Iterators.cpp File Reference

```
#include <array>
#include <cstddef>
#include <iostream>
#include <iterator>
```

Include dependency graph for Iterators.cpp:



Functions

- int main ()

14.19.1 Function Documentation

14.19.1.1 main()

```
int main ( )
```

Iterators

Definition at line 6 of file Iterators.cpp.

```
00007 {
00008     // The type is automatically deduced to std::array<int, 7> (Requires C++17).
00009     // Use the type std::array<int, 7> if your compiler doesn't support C++17.
00010     std::array<int, 7> data{ 0, 1, 2, 3, 4, 5, 6 };
00011     std::size_t length{ std::size(data) };
00012 
00013     // while-loop with explicit index
00014     std::cout « "While loop with explicit index" « std::endl;
00015     std::size_t index{ 0 };
00016     while (index != length)
00017     {
00018         std::cout « data[index] « ' ';
00019         ++index;
00020     }
00021     std::cout « '\n';
00022 
00023     // for-loop with explicit index
00024     std::cout « "For loop with explicit index" « std::endl;
00025     for (index = 0; index < length; ++index)
00026     {
00027         std::cout « data[index] « ' ';
00028     }
00029     std::cout « '\n';
00030 
00031     // for-loop with pointer (Note: ptr can't be const, because we increment it)
00032     std::cout « "For loop with pointer" « std::endl;
00033     for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
00034     {
00035         std::cout « *ptr « ' ';
00036     }
00037     std::cout « '\n';
00038 
00039     // ranged-based for loop
00040     std::cout « "Range based for loop" « std::endl;
00041     for (int i : data)
00042     {
00043         std::cout « i « ' ';
00044     }
00045     std::cout « '\n';
00046 
00047     std::cout « std::endl;
00048 
00050     // Pointers (simplest kind of Iterators)
00051     std::cout « "Iterator: Pointer..." « std::endl;
00052     auto begin{ &data[0] };
00053     // note that this points to one spot beyond the last element
00054     auto end{ begin + std::size(data) };
00055 
00056     // for-loop with pointer
00057     for (auto ptr{ begin }; ptr != end; ++ptr) // ++ to move to next element
00058     {
00059         std::cout « *ptr « ' '; // Indirection to get value of current element
00060     }
00061     std::cout « '\n';
00062 
00063     // Standard library iterators
00064     std::cout « "Standard library iterators..." « std::endl;
00065     // Ask our array for the begin and end points (via the begin and end member functions).
00066     begin = { data.begin() };
00067     end = { data.end() };
00068 
00069     for (auto p{ begin }; p != end; ++p) // ++ to move to next element.
00070     {
00071         std::cout « *p « ' '; // Indirection to get value of current element.
00072     }
00073     std::cout « '\n';
00074 
00075     //
00076     std::cout « "or..." « std::endl;
00077 
00078     begin = { std::begin(data) };
00079     end = { std::end(data) };
00080 
00081     for (auto p{ begin }; p != end; ++p) // ++ to move to next element
00082     {
00083         std::cout « *p « ' '; // Indirection to get value of current element
00084     }
00085     std::cout « '\n';
00086 
00089     return 0;
00090 }
```

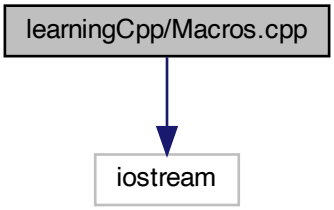
14.20 Iterators.cpp

```
00001 #include <array>
00002 #include <cstddef>
00003 #include <iostream>
00004 #include <iterator>
00005
00006 int main()
00007 {
00008     // The type is automatically deduced to std::array<int, 7> (Requires C++17).
00009     // Use the type std::array<int, 7> if your compiler doesn't support C++17.
00010     std::array<int, 7> data{ 0, 1, 2, 3, 4, 5, 6 };
00011     std::size_t length{ std::size(data) };
00012
00013     // while-loop with explicit index
00014     std::cout « "While loop with explicit index" « std::endl;
00015     std::size_t index{ 0 };
00016     while (index != length)
00017     {
00018         std::cout « data[index] « ' ';
00019         ++index;
00020     }
00021     std::cout « '\n';
00022
00023     // for-loop with explicit index
00024     std::cout « "For loop with explicit index" « std::endl;
00025     for (index = 0; index < length; ++index)
00026     {
00027         std::cout « data[index] « ' ';
00028     }
00029     std::cout « '\n';
00030
00031     // for-loop with pointer (Note: ptr can't be const, because we increment it)
00032     std::cout « "For loop with pointer" « std::endl;
00033     for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
00034     {
00035         std::cout « *ptr « ' ';
00036     }
00037     std::cout « '\n';
00038
00039     // ranged-based for loop
00040     std::cout « "Range based for loop" « std::endl;
00041     for (int i : data)
00042     {
00043         std::cout « i « ' ';
00044     }
00045     std::cout « '\n';
00046
00047     std::cout « std::endl;
00048
00050     // Pointers (simplest kind of Iterators)
00051     std::cout « "Iterator: Pointer..." « std::endl;
00052     auto begin{ &data[0] };
00053     // note that this points to one spot beyond the last element
00054     auto end{ begin + std::size(data) };
00055
00056     // for-loop with pointer
00057     for (auto ptr{ begin }; ptr != end; ++ptr) // ++ to move to next element
00058     {
00059         std::cout « *ptr « ' '; // Indirection to get value of current element
00060     }
00061     std::cout « '\n';
00062
00063     // Standard library iterators
00064     std::cout « "Standard library iterators..." « std::endl;
00065     // Ask our array for the begin and end points (via the begin and end member functions).
00066     begin = { data.begin() };
00067     end = { data.end() };
00068
00069     for (auto p{ begin }; p != end; ++p) // ++ to move to next element.
00070     {
00071         std::cout « *p « ' '; // Indirection to get value of current element.
00072     }
00073     std::cout « '\n';
00074
00075     //
00076     std::cout « "or..." « std::endl;
00077
00078     begin = { std::begin(data) };
00079     end = { std::end(data) };
00080
00081     for (auto p{ begin }; p != end; ++p) // ++ to move to next element
00082     {
00083         std::cout « *p « ' '; // Indirection to get value of current element
00084     }
00085     std::cout « '\n';
00086
```

```
00089     return 0;
00090 }
```

14.21 learningCpp/Macros.cpp File Reference

`#include <iostream>`
Include dependency graph for Macros.cpp:



Macros

- #define PI 3.1415
- #define EULER

Functions

- int main ()

14.21.1 Macro Definition Documentation

14.21.1.1 EULER

`#define EULER`
Definition at line 21 of file Macros.cpp.

14.21.1.2 PI

`#define PI 3.1415`
Header guards (conditional compilation directive)
Definition at line 18 of file Macros.cpp.

14.21.2 Function Documentation

14.21.2.1 main()

`int main ( )`
Definition at line 25 of file Macros.cpp.

```
00025           {
00026
00027 #ifdef PI // or if not defined use #ifndef
```

```
00028     std::cout « "PI is: " « PI « std::endl
00029 //#elif
00030 //#else
00031 #endif
00032
00033 #ifdef EULER
00034     std::cout « "EULER is defined, but not replaceable, or rather replaceable by empty"
00035 #endif
00036
00037     return 0;
00038 }
```

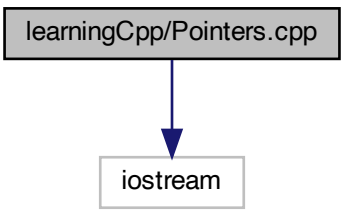
14.22 Macros.cpp

```
00001 //
00002 // Created by Michael Staneker on 01.12.20.
00003 //
00004
00005 #include <iostream>
00006
00008 //#ifnedf SOME_UNIQUE_NAME_HERE
00009 //#define SOME_UNIQUE_NAME_HERE
00010 //
00011 //#endif
00012
00013 // or alternatively use, but bit supported by all compilers
00014 //#pragma once
00017 // define macro (with substitution text)
00018 #define PI 3.1415
00019
00020 // empty substitution text
00021 #define EULER
00022
00023
00024
00025 int main() {
00026
00027 #ifdef PI // or if not defined use #ifndef
00028     std::cout « "PI is: " « PI « std::endl
00029 //#elif
00030 //#else
00031 #endif
00032
00033 #ifdef EULER
00034     std::cout « "EULER is defined, but not replaceable, or rather replaceable by empty"
00035 #endif
00036
00037     return 0;
00038 }
```

14.23 learningCpp/Pointers.cpp File Reference

`#include <iostream>`

Include dependency graph for Pointers.cpp:



Functions

- int main ()

  Brief description.

14.23.1 Function Documentation

14.23.1.1 main()

```
int main ( )
```

Brief description.

14.23.2 Introduction to Pointers

More detailed description

Author

Autor 1

Autor 2

Version

Version number

Date

Date

Precondition

Preconditions ...

Postcondition

Postconditions ...

Bug Bugs ...

Warning

This is a warning ...

Attention

Attenzione Attenzione ...

Note

This is a note

Remarks

This is a remark

Copyright

GNU Public License.

Since

Since when ...

Todo

- add a
- add b
- add c

Test Describing test case ...

User defined paragraph

Contents of the paragraph.

New paragraph under the same heading

Example of a param command with a description consisting of two paragraphs

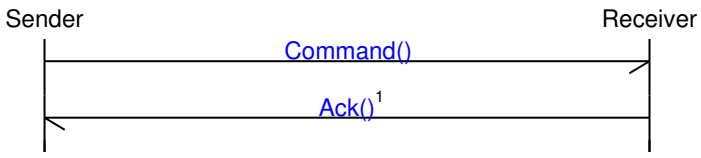
Parameters

p	First paragraph of the param description. Second paragraph of the param description.

Rest of the comment block continues.

```
* Verbatim
* ...
* ...
*
```

The receiver will acknowledge the command by calling Ack().



formula example

$$x_s = \frac{2}{3} \cdot 2^4$$

Some Markdown
See url-refernce: `LearnCpp`
List:

- a
- b
- c

14.23.2.1 Address operator &

14.23.2.2 Indirection operator *

14.23.2.3 Pointers

14.23.2.3.1 pointer

14.23.2.3.2 Pointers

14.23.2.3.3 arithmetic */

```
std::cout « &array[1] « '\n'; // print memory address of array element 1
std::cout « array+1 « '\n'; // print memory address of array pointer + 1
std::cout « array[1] « '\n'; // prints 7
std::cout « *(array+1) « '\n'; // prints 7 (note the parenthesis required here)
```

14.23.2.3.4 memory allocation */

```
//new int; // dynamically allocate an integer (and discard the result)
int *ptr_dyn{ new int }; // dynamically allocate an integer and assign the address to ptr so we can access
      it later
*ptr_dyn = 7;
// equivalent: int *ptr_dyn{ new int { 7 }}
std::cout « "ptr_dyn = " « ptr_dyn « std::endl;
std::cout « "*ptr_dyn = " « *ptr_dyn « std::endl;
// delete
delete ptr_dyn; // return the memory pointed to by ptr to the operating system
ptr_dyn = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)
// Dynamically allocating arrays
int *dyn_array{ new int[5]{ 9, 7, 5, 3, 1 } }; // initialize a dynamic array since C++11
// To prevent writing the type twice, we can use auto. This is often done for types with long names.
//auto *array{ new int[5]{ 9, 7, 5, 3, 1 } };
delete [] dyn_array;
```

14.23.2.3.5 pointers (generic pointer) */

```
int nValue;
float fValue;
struct Something
{
    int n;
    float f;
};
Something sValue;
void *void_ptr;
void_ptr = &nValue; // valid
void_ptr = &fValue; // valid
void_ptr = &sValue; // valid
// ATTENTION: indirection is only possible using a cast
```

14.23.2.3.6 Pointers */

```
int value_for_pointer = 5;
int *primary_ptr = &value_for_pointer;
std::cout « "ptr = " « *primary_ptr « std::endl; // Indirection through pointer to int to get int value
int **ptrptr = &primary_ptr;
std::cout « "ptrptr = " « **ptrptr « std::endl; // first indirection to get pointer to int, second
      indirection to get int value
int **pointer_array = new int*[10]; // allocate an array of 10 int pointers
```

top [("go to the top")]

Definition at line 76 of file Pointers.cpp.

```
00076           {
00077
00078     int x{ 5 };
00079     std::cout « "    x = " « x « '\n'; // print the value of variable x
00080
00082     std::cout « "   &x = " « &x « '\n'; // print the memory address of variable x
00086     std::cout « "*(&x) = " « *(&x) « '\n'; // print the memory address of variable x
00090     //int *iPtr{}; // a pointer to an integer value
00091     //double *dPtr{}; // a pointer to a double value
00092     //int* iPtr2{}; // also valid syntax (acceptable, but not favored)
00093     //int * iPtr3{}; // also valid syntax (but don't do this, it looks like multiplication)
00094     //int *iPtr4{}, *iPtr5{}; // declare two pointers to integer variables (not recommended)
00095
00096     int var{ 5 };
00097     int *ptr{ &var }; // initialize ptr with address of variable v
00098     std::cout « "var = " « var « '\n'; // print the address of variable v
00099     std::cout « "var = " « &var « '\n'; // print the address of variable v
00100     std::cout « "ptr = " « ptr « '\n'; // print the address that ptr is holding
00101     std::cout « "*ptr = " « *ptr « '\n';
00102
00103     //    Pointers are good for:
00104     //    * dynamic arrays
00105     //    * dynamically allocate memory
00106     //    * pass large amount of data to a function (without copying)
00107     //    * pass a function as a parameter to another function
00108     //    * achieve polymorphism when dealing with inheritance
00109     //    * useful for advanced data structures
00113     //assigning it to the literal 0
00114     float *null_ptr { 0 };  // ptr is now a null pointer
00115     float *null_ptr2; // ptr2 is uninitialized
00116     null_ptr2 = 0; // ptr2 is now a null pointer
00117     float *null_ptr3 {nullptr}; // C++11
00121     int array[5]{ 9, 7, 5, 3, 1 };
00122     std::cout « *array « '\n'; // will print 9
00123     int *ptr_for_array{ array };
```

```
00124     std::cout « *ptr_for_array « '\n'; // will print 9
00125
00126     // ARRAYS DECAY INTO POINTERS WHEN PASSED TO FUNCTIONS !!!
00133     std::cout « &array[1] « '\n'; // print memory address of array element 1
00134     std::cout « array+1 « '\n'; // print memory address of array pointer + 1
00135     std::cout « array[1] « '\n'; // prints 7
00136     std::cout « *(array+1) « '\n'; // prints 7 (note the parenthesis required here)
00145     //new int; // dynamically allocate an integer (and discard the result)
00146     int *ptr_dyn{ new int }; // dynamically allocate an integer and assign the address to ptr so we
      can access it later
00147     *ptr_dyn = 7;
00148     // equivalent: int *ptr_dyn{ new int { 7 }}
00149     std::cout « "ptr_dyn = " « ptr_dyn « std::endl;
00150     std::cout « "*ptr_dyn = " « *ptr_dyn « std::endl;
00151
00152     // delete
00153     delete ptr_dyn; // return the memory pointed to by ptr to the operating system
00154     ptr_dyn = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)
00155
00156
00157     // Dynamically allocating arrays
00158     int *dyn_array{ new int[5]{ 9, 7, 5, 3, 1 } }; // initialize a dynamic array since C++11
00159     // To prevent writing the type twice, we can use auto. This is often done for types with long
      names.
00160     //auto *array{ new int[5]{ 9, 7, 5, 3, 1 } };
00161     delete [] dyn_array;
00171     int nValue;
00172     float fValue;
00173     struct Something
00174     {
00175         int n;
00176         float f;
00177     };
00178     Something sValue;
00179     void *void_ptr;
00180     void_ptr = &nValue; // valid
00181     void_ptr = &fValue; // valid
00182     void_ptr = &sValue; // valid
00183     // ATTENTION: indirection is only possible using a cast
00192     int value_for_pointer = 5;
00193
00194     int *primary_ptr = &value_for_pointer;
00195     std::cout « "ptr = " « *primary_ptr « std::endl; // Indirection through pointer to int to get int
      value
00196
00197     int **ptrptr = &primary_ptr;
00198     std::cout « "ptrptr = " « **ptrptr « std::endl; // first indirection to get pointer to int, second
      indirection to get int value
00199
00200     int **pointer_array = new int*[10]; // allocate an array of 10 int pointers
00205     return 0;
00206
00210 }
```

14.24 Pointers.cpp

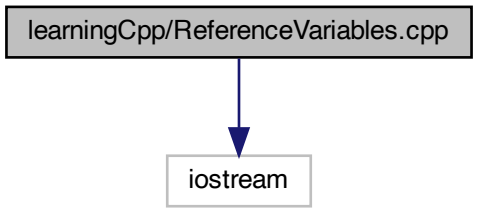
```
00001 //
00002 // Created by Michael Staneker on 03.12.20.
00003 //
00004
00005 #include <iostream>
00006
00076 int main() {
00077
00078     int x{ 5 };
00079     std::cout « "    x = " « x « '\n'; // print the value of variable x
00080
00082     std::cout « "   &x = " « &x « '\n'; // print the memory address of variable x
00086     std::cout « "*(&x) = " « *(&x) « '\n'; // print the memory address of variable x
00090     //int *iPtr{}; // a pointer to an integer value
00091     //double *dPtr{}; // a pointer to a double value
00092     //int* iPtr2{}; // also valid syntax (acceptable, but not favored)
00093     //int * iPtr3{}; // also valid syntax (but don't do this, it looks like multiplication)
00094     //int *iPtr4{}, *iPtr5{}; // declare two pointers to integer variables (not recommended)
00095
00096     int var{ 5 };
00097     int *ptr{ &var }; // initialize ptr with address of variable v
00098     std::cout « "var = " « var « '\n'; // print the address of variable v
00099     std::cout « "var = " « &var « '\n'; // print the address of variable v
00100     std::cout « "ptr = " « ptr « '\n'; // print the address that ptr is holding
00101     std::cout « "*ptr = " « *ptr « '\n';
00102
00103     //    Pointers are good for:
00104     //    * dynamic arrays
00105     //    * dynamically allocate memory
```

```
00106     //    * pass large amount of data to a function (without copying)
00107     //    * pass a function as a parameter to another function
00108     //    * achieve polymorphism when dealing with inheritance
00109     //    * useful for advanced data structures
00113     //assigning it to the literal 0
00114     float *null_ptr { 0 };  // ptr is now a null pointer
00115     float *null_ptr2; // ptr2 is uninitialized
00116     null_ptr2 = 0; // ptr2 is now a null pointer
00117     float *null_ptr3 {nullptr}; // C++11
00121     int array[5]{ 9, 7, 5, 3, 1 };
00122     std::cout « *array « '\n'; // will print 9
00123     int *ptr_for_array{ array };
00124     std::cout « *ptr_for_array « '\n'; // will print 9
00125
00126     // ARRAYS DECAY INTO POINTERS WHEN PASSED TO FUNCTIONS !!!
00133     std::cout « &array[1] « '\n'; // print memory address of array element 1
00134     std::cout « array+1 « '\n'; // print memory address of array pointer + 1
00135     std::cout « array[1] « '\n'; // prints 7
00136     std::cout « *(array+1) « '\n'; // prints 7 (note the parenthesis required here)
00145     //new int; // dynamically allocate an integer (and discard the result)
00146     int *ptr_dyn{ new int }; // dynamically allocate an integer and assign the address to ptr so we
      can access it later
00147     *ptr_dyn = 7;
00148     // equivalent: int *ptr_dyn{ new int { 7 }}
00149     std::cout « "ptr_dyn = " « ptr_dyn « std::endl;
00150     std::cout « "*ptr_dyn = " « *ptr_dyn « std::endl;
00151
00152     // delete
00153     delete ptr_dyn; // return the memory pointed to by ptr to the operating system
00154     ptr_dyn = 0; // set ptr to be a null pointer (use nullptr instead of 0 in C++11)
00155
00156
00157     // Dynamically allocating arrays
00158     int *dyn_array{ new int[5]{ 9, 7, 5, 3, 1 } }; // initialize a dynamic array since C++11
00159     // To prevent writing the type twice, we can use auto. This is often done for types with long
      names.
00160     //auto *array{ new int[5]{ 9, 7, 5, 3, 1 } };
00161     delete [] dyn_array;
00171     int nValue;
00172     float fValue;
00173     struct Something
00174     {
00175         int n;
00176         float f;
00177     };
00178     Something sValue;
00179     void *void_ptr;
00180     void_ptr = &nValue; // valid
00181     void_ptr = &fValue; // valid
00182     void_ptr = &sValue; // valid
00183     // ATTENTION: indirection is only possible using a cast
00192     int value_for_pointer = 5;
00193
00194     int *primary_ptr = &value_for_pointer;
00195     std::cout « "ptr = " « *primary_ptr « std::endl; // Indirection through pointer to int to get int
      value
00196
00197     int **ptrptr = &primary_ptr;
00198     std::cout « "ptrptr = " « **ptrptr « std::endl; // first indirection to get pointer to int, second
      indirection to get int value
00199
00200     int **pointer_array = new int*[10]; // allocate an array of 10 int pointers
00205     return 0;
00206
00210 }
00211
00212
00213
```

14.25 learningCpp/ReferenceVariables.cpp File Reference

```
#include <iostream>
```

Include dependency graph for ReferenceVariables.cpp:



Functions

- int main ()

14.25.1 Function Documentation

14.25.1.1 main()

```
int main ( )
```

Reference variables

Definition at line 7 of file ReferenceVariables.cpp.

```
00007           {
00008
00009     int value {5};
00010
00012     int &reference{ value }; // "reference to" value
00013     //int& reference{ value }; // valid
00014     //int & reference{ value }; // valid
00015
00016     int x{ 5 }; // normal integer
00017     int &y{ x }; // y is a reference to x
00018     int &z{ y }; // z is also a reference to x
00019
00020     std::cout « " x = " «  x « std::endl;
00021     std::cout « " y = " «  y « std::endl;
00022     std::cout « " z = " «  z « std::endl;
00023     std::cout « "&x = " « &x « std::endl;
00024     std::cout « "&y = " « &y « std::endl;
00025     std::cout « "&z = " « &z « std::endl;
00026
00027     // References cannot be reassigned !
00028     // reference = value; // not valid
00029
00030
00033     return 0;
00034 }
```

14.26 ReferenceVariables.cpp

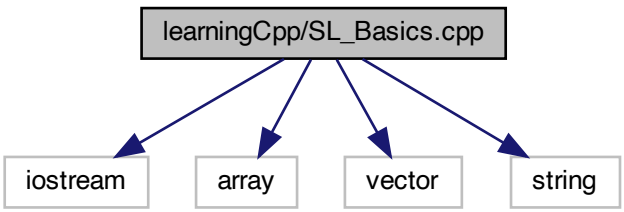
```
00001 //
00002 // Created by Michael Staneker on 03.12.20.
00003 //
00004
00005 #include <iostream>
00006
00007 int main() {
00008
00009     int value {5};
00010
00012     int &reference{ value }; // "reference to" value
00013     //int& reference{ value }; // valid
```

```
00014     //int & reference{ value }; // valid
00015
00016     int x{ 5 }; // normal integer
00017     int &y{ x }; // y is a reference to x
00018     int &z{ y }; // z is also a reference to x
00019
00020     std::cout « " x = " «  x « std::endl;
00021     std::cout « " y = " «  y « std::endl;
00022     std::cout « " z = " «  z « std::endl;
00023     std::cout « "&x = " « &x « std::endl;
00024     std::cout « "&y = " « &y « std::endl;
00025     std::cout « "&z = " « &z « std::endl;
00026
00027     // References cannot be reassigned !
00028     // reference = value; // not valid
00029
00030
00033     return 0;
00034 }
00035
```

14.27 learningCpp/SL_Basics.cpp File Reference

```
#include <iostream>
#include <array>
#include <vector>
#include <string>
```

Include dependency graph for SL_Basics.cpp:



Functions

- int main ()

14.27.1 Function Documentation

14.27.1.1 main()

```
int main ( )
```

14.27.2 Introduction to the standard library

14.27.2.1 std::array

14.27.2.2 std::vector

14.27.2.3 std::string

14.27.2.4 Algorithms

- **Inspectors** are used to view (not modify) data in container (including searching and counting)
- **Mutators** are used to modify data in a container (including sorting and shuffling)
- **Facilitators** are used to generate a result based on values of the data members

Definition at line 14 of file SL_Basics.cpp.

```
00014           {
00016     //std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
00017     std::array<int, 5> my_array{9, 7, 5, 3, 1}; // list initialization
00018     my_array[0] = 10; // standard accessing
00019     my_array.at(1) = 8; // other possibility
00020     std::cout « "size of my_array: " « my_array.size() « std::endl;
00024     // dynamic arrays without the need of dynamically allocating memory
00025
00026     //std::vector<int> vec_array;
00027     //std::vector<int> vec_array = { 9, 7, 5, 3, 1 }; // use initializer list to initialize array
      (Before C++11)
00028     std::vector<int> vec_array { 9, 7, 5, 3, 1 }; // use uniform initialization to initialize array
00029     vec_array[0] = 10; // standard accessing
00030     vec_array.at(1) = 8; // other possibility
00031     std::cout « "size of vec_array: " « vec_array.size() « std::endl;
00032     // resize
00033     vec_array.resize(10);
00034     std::cout « "size of vec_array (after resize): " « vec_array.size() « std::endl;
00057     return 0;
00058 }
```

14.28 SL_Basics.cpp

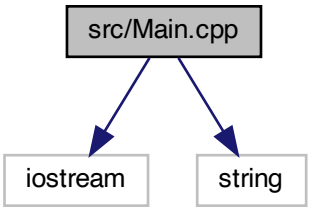
```
00001 //
00002 // Created by Michael Staneker on 03.12.20.
00003 //
00004
00005 #include <iostream>
00006
00007 #include <array> // C++ built in fixed arrays in a safer and more usable form
00008 #include <vector> // makes working with dynamic arrays safer and easier
00009 #include <string> //TODO: section about std::string
00014 int main() {
00016     //std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
00017     std::array<int, 5> my_array{9, 7, 5, 3, 1}; // list initialization
00018     my_array[0] = 10; // standard accessing
00019     my_array.at(1) = 8; // other possibility
00020     std::cout « "size of my_array: " « my_array.size() « std::endl;
00024     // dynamic arrays without the need of dynamically allocating memory
00025
00026     //std::vector<int> vec_array;
00027     //std::vector<int> vec_array = { 9, 7, 5, 3, 1 }; // use initializer list to initialize array
      (Before C++11)
00028     std::vector<int> vec_array { 9, 7, 5, 3, 1 }; // use uniform initialization to initialize array
00029     vec_array[0] = 10; // standard accessing
00030     vec_array.at(1) = 8; // other possibility
00031     std::cout « "size of vec_array: " « vec_array.size() « std::endl;
00032     // resize
00033     vec_array.resize(10);
00034     std::cout « "size of vec_array (after resize): " « vec_array.size() « std::endl;
00057     return 0;
00058 }
```

14.29 README.md File Reference

14.30 src/Main.cpp File Reference

```
#include <iostream>
#include <string>
```

Include dependency graph for Main.cpp:



Functions

- int main ()

14.30.1 Function Documentation

14.30.1.1 main()

```
int main ( )
```

Definition at line 4 of file Main.cpp.

```
00004            {
00005
00006     printf("Hello World!\n");
00007
00008     return 0;
00009 }
```

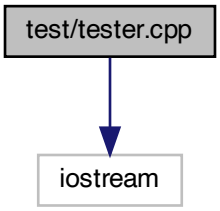
14.31 Main.cpp

```
00001 #include <iostream>
00002 #include <string>
00003
00004 int main() {
00005
00006     printf("Hello World!\n");
00007
00008     return 0;
00009 }
00010
```

14.32 test/tester.cpp File Reference

```
#include <iostream>
```

Include dependency graph for tester.cpp:



Functions

- int main ()

14.32.1 Function Documentation

14.32.1.1 main()

```
int main ( )
```

Definition at line 3 of file tester.cpp.

```
00003            {
00004
00005     std::cout « "This is a tester file" « std::endl;
00006     return 0;
00007
00008 }
```

14.33 tester.cpp

```
00001 #include <iostream>
00002
00003 int main() {
00004
00005     std::cout « "This is a tester file" « std::endl;
00006     return 0;
00007
00008 }
00009
```