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

Hierarchical Index

10.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 11

Class Index

11.1 Class List

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

Chapter 12

File Index

12.1 File List

Here is a list of all files with brief descriptions:

include/ConceptClass.h ??
learningCpp/Basics/Basics.cpp ??
learningCpp/Basics/BitManipulation.cpp ??
learningCpp/Basics/CommandLineArguments.cpp ??
learningCpp/Basics/Functions.cpp ??
learningCpp/Basics/Functions.h ??
learningCpp/Basics/Iterators.cpp ??
learningCpp/Basics/Macros.cpp ??
learningCpp/Basics/Pointers.cpp ??
learningCpp/Basics/ReferenceVariables.cpp ??
learningCpp/Errors/ErrorHandling.cpp ??
learningCpp/Errors/ErrorHandling.h ??
learningCpp/Errors/Exceptions.cpp ??
learningCpp/Errors/Exceptions.h ??
learningCpp/Errors/TestingExceptions.cpp ??
learningCpp/OOP/ConceptClass.cpp ??
learningCpp/OOP/constants.h ??
learningCpp/OOP/Inheritance.cpp ??
learningCpp/OOP/Inheritance.h ??
learningCpp/OOP/SampleClass.cpp ??
learningCpp/OOP/SampleClass.h ??
learningCpp/OOP/TemplateClass.h ??
learningCpp/OOP/TestInheritance.cpp ??
learningCpp/OOP/TestSampleClass.cpp ??
learningCpp/OOP/TestTemplateClass.cpp ??
learningCpp/OOP/Timer.cpp ??
learningCpp/OOP/Timer.h ??
learningCpp/StandardLibrary/SL_Basics.cpp ??
src/Main.cpp ??
test/tester.cpp ??

Chapter 13

Namespace Documentation

13.1 constants Namespace Reference

Variables

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

13.1.1 Variable Documentation

13.1.1.1 avogadro

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

Definition at line 11 of file constants.h.

13.1.1.2 pi

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

Definition at line 10 of file constants.h.

Chapter 14

Class Documentation

14.1 Array< T > Class Template Reference

```
#include "TemplateClass.h"
```

Public Member Functions

- Array (int length)
- Array (const Array &)=delete
- Array & operator= (const Array &)=delete
- ~Array ()
- void Erase ()
- T & operator[ ] (int index)
- int getLength () const
- void print ()

Private Attributes

- int m_length {}
- T ∗ m_data {}

14.1.1 Detailed Description

template< class T>
class Array< T >

14.1.2 Class templates

In order to create classes for different data types use **template classes**.

14.1.2.1 Specialization

14.1.2.1.1 Function specialization It is possible to overwrite (individual) member function for

- different data types
- pointer (types)
- ...

14.1.2.1.2 Class specialization It is possible to overwrite an entire template class for specific data types.
See `class specialization` for reference.
Definition at line 36 of file TemplateClass.h.

14.1.3 Constructor & Destructor Documentation

14.1.3.1 Array() [1/2]

```
template<class T >
Array< T >::Array (
             int length )
```

14.1.3.2 Array() [2/2]

```
template<class T >
Array< T >::Array (
             const Array< T > &  )  [delete]
```

14.1.3.3 ∼Array()

```
template<class T >
Array< T >::∼Array ( )
```

14.1.4 Member Function Documentation

14.1.4.1 Erase()

```
template<class T >
void Array< T >::Erase ( )
```

14.1.4.2 getLength()

```
template<class T >
int Array< T >::getLength ( ) const
```

14.1.4.3 operator=()

```
template<class T >
Array& Array< T >::operator= (
             const Array< T > &  )  [delete]
```

14.1.4.4 operator[]()

```
template<class T >
T& Array< T >::operator[] (
             int index )
```

14.1.4.5 print()

```
template<class T >
void Array< T >::print ( )
```

14.1.5 Member Data Documentation

14.1.5.1 m_data

```
template<class T >
T* Array< T >::m_data {}  [private]
```

Definition at line 40 of file TemplateClass.h.

14.1.5.2 m_length

```
template<class T >
int Array< T >::m_length {}  [private]
```

Definition at line 39 of file TemplateClass.h.

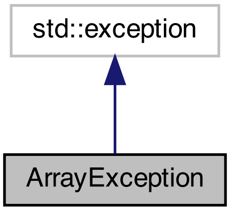
The documentation for this class was generated from the following file:

- learningCpp/OOP/TemplateClass.h

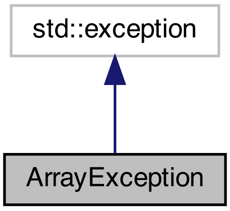
14.2 ArrayException Class Reference

```
#include "Exceptions.h"
```

Inheritance diagram for ArrayException:



Collaboration diagram for ArrayException:



Public Member Functions

- ArrayException (std::string_view error)
- const char ∗ what () const noexcept override

Private Attributes

- std::string m_error {}

14.2.1 Detailed Description

14.2.2 Exceptions

Exceptions in C++ are implemented using three keywords that work in conjunction with each other:

- throw
- try
- catch

Exception handling is best used when all of the following are true:

- the error being handled is likely to occur only infrequently.
- the error is serious and execution could not continue otherwise.
- the error cannot be handled at the place where it occurs.
- there isn’t a good alternative way to return an error code back to the caller.

14.2.2.1 Throwing exceptions

A throw statement is used to signal that an exception or error case has occurred, e.g.:

```
 {C++}
throw -1; // throw a literal integer value
throw ENUM_INVALID_INDEX; // throw an enum value
throw "Can not take square root of negative number"; // throw a literal C-style (const char*) string
throw dX; // throw a double variable that was previously defined
throw MyException("Fatal Error"); // Throw an object of class MyException
```

14.2.2.2 Try blocks

Try blocks act as observers, looking for any exceptions that are thrown within the block, e.g.:

```
 {C++}
try
{
// Statements that may throw exceptions you want to handle go here
throw -1; // here’s a trivial throw statement
}
```

14.2.2.3 Handling exceptions

Actually handling exceptions is the job of the catch block(s). The catch keyword is used to define a block of code (called a catch block) that handles exceptions for a single data type, e.g.:

```
 {C++}
catch (int x)
{
// Handle an exception of type int here
std::cerr « "We caught an int exception with value" « x « ’\n’;
}
```

14.2.2.4 Throwing exceptions outside a try-block

...

14.2.2.5 Catch all handler

To catch uncaught exceptions, not regarding the type of exception:

```
 {C++}
catch (...) // catch-all handler
{
    std::cout « "We caught an exception of an undetermined type\n";
}
```

14.2.2.6 Exception classes

...

14.2.2.7 std::exception

Many of the classes and operators in the standard library throw exception classes on failure. For example, operator new can throw std::bad_alloc if it is unable to allocate enough memory. A failed dynamic_cast will throw std::bad↩_cast. And so on. As of C++17, there are 25 different exception classes that can be thrown, with more being added in each subsequent language standard.
It is possible to extend std::exception, by inheriting.

14.2.2.8 Rethrowing

When rethrowing the same exception, use the *throw* keyword by itself.

14.2.2.9 noexcept

See `exception specifier`.
It is possible to declare functions non-throwing using the **noexcept** specifier.
Definition at line 105 of file Exceptions.h.

14.2.3 Constructor & Destructor Documentation

14.2.3.1 ArrayException()

```
ArrayException::ArrayException (
               std::string_view error )
```

Definition at line 8 of file Exceptions.cpp.

```
00009          : m_error{error}
00010 {
00011 }
```

14.2.4 Member Function Documentation

14.2.4.1 what()

```
const char* ArrayException::what ( ) const  [inline], [override], [noexcept]
```

Definition at line 113 of file Exceptions.h.

```
00113                                             {
00114         return m_error.c_str();
00115     }
```

14.2.5 Member Data Documentation

14.2.5.1 m_error

```
std::string ArrayException::m_error {}  [private]
```

Definition at line 108 of file Exceptions.h.

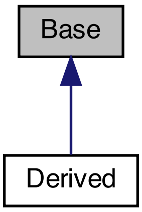
The documentation for this class was generated from the following files:

- learningCpp/Errors/Exceptions.h
- learningCpp/Errors/Exceptions.cpp

14.3 Base Class Reference

```
#include "Inheritance.h"
```

Inheritance diagram for Base:



Public Member Functions

- Base (int id=0, int var_private=0, int var_protected=0, int var_public=0)
- int getId () const
- virtual void print ()

Public Attributes

- int m_public

Protected Member Functions

- int getPrivate () const

Protected Attributes

- int m_protected

Private Attributes

- int m_id
- int m_private

14.3.1 Detailed Description

14.3.2 Inheritance

14.3.2.1 Access specifiers

There are three access specifiers

- **public**: accessible from base and derived class, and from outside
- **protected**: accessible from base class and derived class
- **private**: accessible from base class

```
 {C++}
class Derived: <access specifier> Base
{
}
```

14.3.2.1.1 Public inheritance

- public --> public
- protected --> protected
- private --> inaccessible

14.3.2.1.2 Protected inheritance

- public --> protected
- protected --> protected
- private --> inaccessible

14.3.2.1.3 Private Inheritance

- public --> private
- protected --> private
- private --> inaccessible

14.3.2.2 Multiple inheritance

C++ supports **multiple inheritance**, but many problems can occur. Since most of the problems solvable with multiple inheritance can be solved without multiple inheritance, prefer solutions without multiple inheritance.
Avoid multiple inheritance unless alternatives lead to more complexity.

14.3.2.3 Virtual functions and Polymorphism

A **virtual function** is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class. This capability is known as **polymorphism**.
Attention: Resolving a virtual function call takes longer than resolving a regular one. Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions.
When dealing with inheritance, (overwritten) **destructors** should always be virtual!

14.3.2.4 Override and final specifiers

To help address the issue of functions that are meant to be overrides but aren't, C++11 introduced the **override** specifier. The override specifier can be applied to any override function by placing the specifier in the same place const would go.
There may be cases where you don't want someone to be able to override a virtual function, or inherit from a class. The **final** specifier can be used to tell the compiler to enforce this. If the user tries to override a function or inherit from a class that has been specified as final, the compiler will give a compile error.

14.3.2.5 Pure virtual functions, abstract base classes and interface classes

14.3.2.5.1 Pure virtual functions C++ allows to create a special kind of virtual function called a pure virtual function (or abstract function) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.
Any class with at least one pure virtual function becomes an abstract base class and cannot be instantiated

14.3.2.5.2 Abstract base classes

Abstract base classes can not be instantiated!

14.3.2.5.3 Interface classes

An **interface class** has no member variables and only pure virtual (member) functions. Thus, interface classes are pure definitions and have no actual implementations.
Definition at line 99 of file Inheritance.h.

14.3.3 Constructor & Destructor Documentation

14.3.3.1 Base()

```
Base::Base (
              int id = 0,
              int var_private = 0,
              int var_protected = 0,
              int var_public = 0 )
```

Definition at line 7 of file Inheritance.cpp.

```
00008          : m_id{ id }, m_private{ var_private }, m_protected{ var_protected },
00009            m_public{ var_public }
00010 {
00011     std::cout « "Base constructor called ..." « std::endl;
00012 }
```

14.3.4 Member Function Documentation

14.3.4.1 getId()

```
int Base::getId ( ) const
```

Definition at line 14 of file Inheritance.cpp.

```
00014                        {
00015     return m_id;
00016 }
```

14.3.4.2 getPrivate()

```
int Base::getPrivate ( ) const  [protected]
```

Definition at line 18 of file Inheritance.cpp.

```
00018                             {
00019     std::cout « "getPrivate() from Base" « std::endl;
00020     return m_private;
00021 }
```

14.3.4.3 print()

```
void Base::print ( )  [virtual]
```

Reimplemented in Derived.

Definition at line 23 of file Inheritance.cpp.

```
00023                    {
00024     std::cout « "Print from Base class!" « std::endl;
00025 }
```

14.3.5 Member Data Documentation

14.3.5.1 m_id

```
int Base::m_id  [private]
```

Definition at line 102 of file Inheritance.h.

14.3.5.2 m_private

```
int Base::m_private  [private]
```

Definition at line 103 of file Inheritance.h.

14.3.5.3 m_protected

```
int Base::m_protected  [protected]
```

Definition at line 105 of file Inheritance.h.

14.3.5.4 m_public

```
int Base::m_public
```

Definition at line 108 of file Inheritance.h.

The documentation for this class was generated from the following files:

- learningCpp/OOP/Inheritance.h
- learningCpp/OOP/Inheritance.cpp

14.4 ConceptClass Class Reference

```
#include "ConceptClass.h"
```

Public Member Functions

- ConceptClass (int a, int b)

Public Attributes

- int member_a
- int member_b

14.4.1 Detailed Description

Definition at line 12 of file ConceptClass.h.

14.4.2 Constructor & Destructor Documentation

14.4.2.1 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 }
```

14.4.3 Member Data Documentation

14.4.3.1 member_a

`int ConceptClass::member_a`

Parameters

member	a

Definition at line 22 of file ConceptClass.h.

14.4.3.2 member_b

`int ConceptClass::member_b`

Parameters

member	b

Definition at line 24 of file ConceptClass.h.

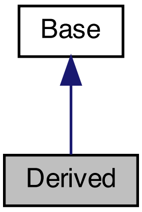
The documentation for this class was generated from the following files:

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

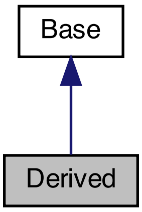
14.5 Derived Class Reference

`#include "Inheritance.h"`

Inheritance diagram for Derived:



Collaboration diagram for Derived:



Public Member Functions

- Derived (double cost=0.0, int id=0, int var_private=0, int var_protected=0, int var_public=0)
- double getCost () const
- double getProtected () const
- double getPrivate () const
- virtual void print ()

Private Attributes

- double m_cost

Additional Inherited Members

14.5.1 Detailed Description

Definition at line 117 of file Inheritance.h.

14.5.2 Constructor & Destructor Documentation

14.5.2.1 Derived()

```
Derived::Derived (
              double cost = 0.0,
              int id = 0,
              int var_private = 0,
              int var_protected = 0,
              int var_public = 0 )
```

Definition at line 29 of file Inheritance.cpp.

```
00030         : Base{ id, var_private, var_protected, var_public }, // Call Base(int) constructor with value id!
00031           m_cost{ cost }
00032 {
00033     std::cout « "Derived constructor called ..." « std::endl;
00034 }
```

14.5.3 Member Function Documentation

14.5.3.1 getCost()

```
double Derived::getCost ( ) const
```

Definition at line 36 of file Inheritance.cpp.

```
00036                               {
00037     return m_cost;
00038 }
```

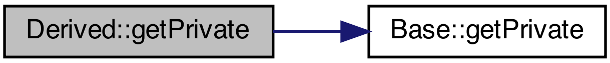
14.5.3.2 getPrivate()

```
double Derived::getPrivate ( ) const
```

Definition at line 44 of file Inheritance.cpp.

```
00044                                  {
00045     std::cout « "getPrivate() from Derived" « std::endl;
00046     return Base::getPrivate();
00047 }
```

Here is the call graph for this function:



14.5.3.3 getProtected()

```
double Derived::getProtected ( ) const
```

Definition at line 40 of file Inheritance.cpp.

```
00040                                    {
00041     return m_protected;
00042 }
```

14.5.3.4 print()

```
void Derived::print ( ) [virtual]
```

Reimplemented from Base.

Definition at line 49 of file Inheritance.cpp.

```
00049                      {
00050     std::cout « "Print from Derived class!" « std::endl;
00051 }
```

14.5.4 Member Data Documentation

14.5.4.1 m_cost

```
double Derived::m_cost [private]
```

Definition at line 120 of file Inheritance.h.

The documentation for this class was generated from the following files:

- learningCpp/OOP/Inheritance.h
- learningCpp/OOP/Inheritance.cpp

14.6 Exceptions Class Reference

```
#include "Exceptions.h"
```

14.6.1 Detailed Description

Definition at line 139 of file Exceptions.h.
The documentation for this class was generated from the following file:

- learningCpp/Errors/Exceptions.h

14.7 IntArray Class Reference

```
#include "Exceptions.h"
```

Public Member Functions

- IntArray ()
- int getLength () const
- int & operator[ ] (const int index)

Private Attributes

- int m_data [3]

14.7.1 Detailed Description

Definition at line 118 of file Exceptions.h.

14.7.2 Constructor & Destructor Documentation

14.7.2.1 IntArray()

```
IntArray::IntArray ( )  [inline]
```

Definition at line 125 of file Exceptions.h.

```
00125 {}
```

14.7.3 Member Function Documentation

14.7.3.1 getLength()

```
int IntArray::getLength ( ) const
```

Definition at line 15 of file Exceptions.cpp.

```
00015                                   {
00016     return 3;
00017 }
```

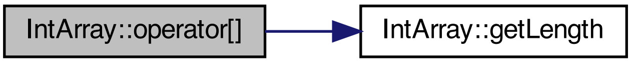
14.7.3.2 operator[]()

```
int & IntArray::operator[] (
             const int index )
```

Definition at line 19 of file Exceptions.cpp.

```
00020 {
00021     if (index < 0 || index >= getLength())
00022         throw ArrayException("Invalid index");
00023
00024     return m_data[index];
00025 }
```

Here is the call graph for this function:



14.7.4 Member Data Documentation

14.7.4.1 m_data

`int IntArray::m_data[3] [private]`

Definition at line 122 of file Exceptions.h.

The documentation for this class was generated from the following files:

- learningCpp/Errors/Exceptions.h
- learningCpp/Errors/Exceptions.cpp

14.8 SampleClass Class Reference

`#include "SampleClass.h"`

Public Types

- enum FruitType { APPLE, BANANA, CHERRY }

Public Member Functions

- int get_member_a ()
- int get_member_b ()
- void set_member_a (int a)
- void set_member_b (int b)
- void set_members_using_this (int member_a, int member_b)
- void const_member_function () const
- SampleClass ()
- SampleClass (int a, int b=0)
- SampleClass (int a, int b, int c)
- SampleClass (const SampleClass &sample_class)
- ∼SampleClass ()
- int operator() (int i)

Static Public Member Functions

- static void static_member_function ()

Static Public Attributes

- static int static_member_variable = 5

Private Attributes

- int member_a { 0 }
- int member_b { 0 }

Friends

- void friend_function (SampleClass &sample_class)
- SampleClass operator+ (const SampleClass &s_1, const SampleClass &s_2)
- std::ostream & operator<< (std::ostream &out, const SampleClass &sample_class)

14.8.1 Detailed Description

14.8.2 Classes - OOP

Use structs for data-only objects and classes otherwise!

14.8.2.1 Properties

- member variables are private per default
  - public in case of structs
- if no constructor is given, a default constructor is created
- getter should either return by value or reference
- const class objects can only call const member functions

14.8.2.2 Friend functions and classes

See `Friend functions and classes` for reference!
Classes keep your data private and encapsulated. However, in some situations you need to have classes and functions outside of those classes that need to work closely together.
For doing this, without exposing the function use the **friend** identifier.
It is possible to have

- friend functions
- friend member functions
- friend classes
- ...

14.8.2.3 Overloading operators

`Overloading assignment operator`

14.8.2.4 Shallow vs. deep copy

The default copy mechanism for classes is **memberwise** copy (also called **shallow copy**), which works for simple classes, without dynamically reserved memory, very good.
However, a **deep copy** allocates memory for the copy and then copies the actual value, so that the copy lives in distinct memory from the source. This requires to write copy constructors and overloaded assignment operators.

- The default copy constructor and default assignment operators do shallow copies, which is fine for classes that contain no dynamically allocated variables.
- Classes with dynamically allocated variables need to have a copy constructor and assignment operator that do a deep copy.
- Favor using classes in the standard library over doing your own memory management.

14.8.2.5 Object relations

Property/type	Composition	Composition	Composition	Composition
relationship	whole/part	whole/part	unrelated	unrelated
members belong to multiple classes	No	Yes	Yes	Yes
members existence managed by class	Yes	No	No	No
directionality	Uni	Uni	Uni or bi	Uni
relationship verb	part-of	has-a	uses-a	depends-on

14.8.2.5.1 Composition To qualify as a composition, an object and a part must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can only belong to one object (class) at a time
- The part (member) has its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

Therefore:

- Typically use normal member variables
- Can use pointer members if the class handles object allocation/deallocation itself
- Responsible for creation/destruction of parts

14.8.2.5.2 Aggregation To qualify as an aggregation, a whole object and its parts must have the following relationship:

- The part (member) is part of the object (class)
- The part (member) can belong to more than one object (class) at a time
- The part (member) does not have its existence managed by the object (class)
- The part (member) does not know about the existence of the object (class)

Therefore:

- Typically use pointer or reference members that point to or reference objects that live outside the scope of the aggregate class
- Not responsible for creating/destroying parts

14.8.2.5.3 Association

- To qualify as an association, an object and another object must have the following relationship:
- The associated object (member) is otherwise unrelated to the object (class)
- The associated object (member) can belong to more than one object (class) at a time
- The associated object (member) does not have its existence managed by the object (class)∗
- The associated object (member) may or may not know about the existence of the object (class)

14.8.2.6 Container classes

See `Container classes`
Container classes typically implement a fairly standardized minimal set of functionality. Most well-defined containers will include functions that:

- Create an empty container (via a constructor)
- Insert a new object into the container
- Remove an object from the container
- Report the number of objects currently in the container
- Empty the container of all objects
- Provide access to the stored objects
- Sort the elements (optional)

Definition at line 128 of file SampleClass.h.

14.8.3 Member Enumeration Documentation

14.8.3.1 FruitType

```
enum SampleClass::FruitType
```

Enumerator

APPLE	
BANANA	
CHERRY	

Definition at line 172 of file SampleClass.h.

```
00172                     {
00173         APPLE,
00174         BANANA,
00175         CHERRY
00176     };
```

14.8.4 Constructor & Destructor Documentation

14.8.4.1 SampleClass() [1/4]

```
SampleClass::SampleClass ( )
```

Definition at line 7 of file SampleClass.cpp.

```
00007                          {
00008     std::cout « "Default constructor was called ..." « std::endl;
00009     member_a = 0;
00010     member_b = 0;
00011 }
```

14.8.4.2 SampleClass() [2/4]

```
SampleClass::SampleClass (
            int a,
            int b = 0 )
```

Definition at line 21 of file SampleClass.cpp.

```
00022 : member_a{ a }, member_b{ b }
```

```
00023 {
00024     std::cout « "Constructor: SampleClass(" « a « ", " « b « ") ..." « std::endl;
00025 }
```

14.8.4.3 SampleClass() [3/4]

```
SampleClass::SampleClass (
             int a,
             int b,
             int c )
```

Definition at line 27 of file SampleClass.cpp.

```
00027                                            : SampleClass{ a, b } {
00028     std::cout « "Constructor: SampleClass(" « a « ", " « b « ", " « c « ") ..." « std::endl;
00029 }
```

14.8.4.4 SampleClass() [4/4]

```
SampleClass::SampleClass (
             const SampleClass & sample_class )
```

Definition at line 36 of file SampleClass.cpp.

```
00036                                                         :
00037         member_a(sample_class.member_a), member_b(sample_class.member_b)
00038 {
00039     std::cout « "Copy constructor called\n"; // just to prove it works
00040 }
```

14.8.4.5 ∼SampleClass()

```
SampleClass::∼SampleClass ( )
```

Definition at line 32 of file SampleClass.cpp.

```
00032                            {
00033     std::cout « "Destructor was called" « std::endl;
00034 }
```

14.8.5 Member Function Documentation

14.8.5.1 const_member_function()

```
void SampleClass::const_member_function ( ) const
```

Definition at line 65 of file SampleClass.cpp.

```
00065                                               {
00066     std::cout « "This is a const member function!" « std::endl;
00067 }
```

14.8.5.2 get_member_a()

```
int SampleClass::get_member_a ( )
```

Definition at line 42 of file SampleClass.cpp.

```
00042                                {
00043     return member_a;
00044 }
```

14.8.5.3 get_member_b()

```
int SampleClass::get_member_b ( )
```

Definition at line 46 of file SampleClass.cpp.

```
00046                                {
00047     return member_b;
00048 }
```

14.8.5.4 operator()()

```
int SampleClass::operator() (
            int i )
```

Definition at line 92 of file SampleClass.cpp.

```
00092                              {
00093     return (member_a += i);
00094 }
```

14.8.5.5 set_member_a()

```
void SampleClass::set_member_a (
            int a )
```

Definition at line 50 of file SampleClass.cpp.

```
00050                                 {
00051     std::cout « "set member_a to: " « a « std::endl;
00052     member_a = a;
00053 }
```

14.8.5.6 set_member_b()

```
void SampleClass::set_member_b (
            int b )
```

Definition at line 55 of file SampleClass.cpp.

```
00055                                 {
00056     std::cout « "set member_b to: " « b « std::endl;
00057     member_b = b;
00058 }
```

14.8.5.7 set_members_using_this()

```
void SampleClass::set_members_using_this (
            int member_a,
            int member_b )
```

Definition at line 60 of file SampleClass.cpp.

```
00060                                                                 {
00061     this->member_a = member_a;
00062     this->member_b = member_b;
00063 }
```

14.8.5.8 static_member_function()

```
void SampleClass::static_member_function ( )  [static]
```

Definition at line 69 of file SampleClass.cpp.

```
00069                                     {
00070     std::cout « "This is a static member function" « std::endl;
00071 }
```

14.8.6 Friends And Related Function Documentation

14.8.6.1 friend_function

```
void friend_function (
            SampleClass & sample_class )  [friend]
```

Definition at line 73 of file SampleClass.cpp.

```
00073                                             {
00074     std::cout « "This is a friend function" « std::endl;
00075     std::cout « "Accessing private member member_a: " « sample_class.member_a « std::endl;
00076 }
```

14.8.6.2 operator+

```
SampleClass operator+ (
             const SampleClass & s_1,
             const SampleClass & s_2 )  [friend]
```

Definition at line 78 of file SampleClass.cpp.

```
00078                                                                              {
00079     std::cout « "overloaded operator+ for SampleClass!" « std::endl;
00080     return SampleClass(s_1.member_a + s_2.member_a, s_1.member_b + s_2.member_b);
00081 }
```

14.8.6.3 operator<<

```
std::ostream& operator<< (
             std::ostream & out,
             const SampleClass & sample_class )  [friend]
```

Definition at line 83 of file SampleClass.cpp.

```
00083                                                                                  {
00084
00085     out « std::endl
00086         « "member_a = " « sample_class.member_a « std::endl
00087         « "member_b = " « sample_class.member_b « std::endl;
00088
00089     return out;
00090 }
```

14.8.7 Member Data Documentation

14.8.7.1 member_a

```
int SampleClass::member_a { 0 }  [private]
```

Definition at line 130 of file SampleClass.h.

14.8.7.2 member_b

```
int SampleClass::member_b { 0 }  [private]
```

Definition at line 131 of file SampleClass.h.

14.8.7.3 static_member_variable

```
int SampleClass::static_member_variable = 5  [static]
```

Definition at line 134 of file SampleClass.h.

The documentation for this class was generated from the following files:

- learningCpp/OOP/SampleClass.h
- learningCpp/OOP/SampleClass.cpp

14.9 SQRT Class Reference

```
#include "Exceptions.h"
```

Static Public Member Functions

- static double mySqrt (double x)

14.9.1 Detailed Description

Definition at line 133 of file Exceptions.h.

14.9.2 Member Function Documentation

14.9.2.1 mySqrt()

```
double SQRT::mySqrt (
            double x )  [static]
```

Definition at line 28 of file Exceptions.cpp.

```
00028                         {
00029     // If the user entered a negative number, this is an error condition
00030     if (x < 0.0)
00031         throw "Can not take sqrt of negative number"; // throw exception of type const char*
00032 
00033     return sqrt(x);
00034 }
```

The documentation for this class was generated from the following files:

- learningCpp/Errors/Exceptions.h
- learningCpp/Errors/Exceptions.cpp

14.10 StaticArray< T, size > Class Template Reference

```
#include "TemplateClass.h"
```

Public Member Functions

- T ∗ getArray ()
- T & operator[ ] (int index)

Private Attributes

- T m_array [size]

14.10.1 Detailed Description

template<class T, int size>
class StaticArray< T, size >

Definition at line 63 of file TemplateClass.h.

14.10.2 Member Function Documentation

14.10.2.1 getArray()

```
template<class T , int size>
T* StaticArray< T, size >::getArray ( )
```

14.10.2.2 operator[]()

```
template<class T , int size>
T& StaticArray< T, size >::operator[ ] (
            int index )
```

14.10.3 Member Data Documentation

14.10.3.1 m_array

```
template<class T , int size>
T StaticArray< T, size >::m_array[size]  [private]
```

Definition at line 67 of file TemplateClass.h.
The documentation for this class was generated from the following file:

- learningCpp/OOP/TemplateClass.h

14.11 Timer Class Reference

```
#include "Timer.h"
```

Public Member Functions

- Timer ()
- void reset ()
- double elapsed () const

Private Types

- using clock_t = std::chrono::high_resolution_clock
- using second_t = std::chrono::duration< double, std::ratio< 1 > >

Private Attributes

- std::chrono::time_point< clock_t > m_beg

14.11.1 Detailed Description

Definition at line 11 of file Timer.h.

14.11.2 Member Typedef Documentation

14.11.2.1 clock_t

```
using Timer::clock_t = std::chrono::high_resolution_clock  [private]
```

Definition at line 14 of file Timer.h.

14.11.2.2 second_t

```
using Timer::second_t = std::chrono::duration<double, std::ratio<1> >  [private]
```

Definition at line 15 of file Timer.h.

14.11.3 Constructor & Destructor Documentation

14.11.3.1 Timer()

```
Timer::Timer ( )
```

Definition at line 7 of file Timer.cpp.

```
00007               : m_beg(clock_t::now())
00008 {
00009 }
```

14.11.4 Member Function Documentation

14.11.4.1 elapsed()

```
double Timer::elapsed ( ) const
```

Definition at line 16 of file Timer.cpp.

```
{
    return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
}
```

14.11.4.2 reset()

```
void Timer::reset ( )
```

Definition at line 11 of file Timer.cpp.

```
{
    m_beg = clock_t::now();
}
```

14.11.5 Member Data Documentation

14.11.5.1 m_beg

```
std::chrono::time_point<clock_t> Timer::m_beg [private]
```

Definition at line 17 of file Timer.h.

The documentation for this class was generated from the following files:

- learningCpp/OOP/Timer.h
- learningCpp/OOP/Timer.cpp

Chapter 15

File Documentation

15.1 documents/CMakeIntroduction.md File Reference

15.2 documents/Markdown.md File Reference

15.3 documents/structure.md File Reference

15.4 documents/Unit-Tests.md File Reference

15.5 include/ConceptClass.h File Reference

Classes

- class ConceptClass

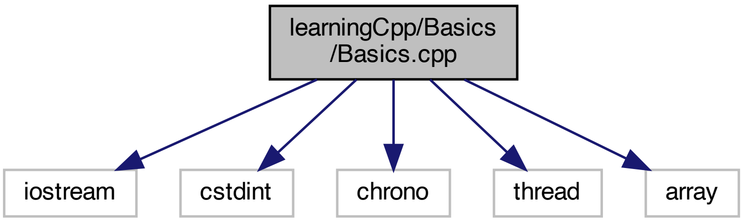
15.6 ConceptClass.h

```
#ifndef CPP_CONCEPTS_PROJECT_CONCEPTCLASS_H
#define CPP_CONCEPTS_PROJECT_CONCEPTCLASS_H

class ConceptClass {
public:
    ConceptClass(int a,int b);

    int member_a;
    int member_b;
};


#endif //CPP_CONCEPTS_PROJECT_CONCEPTCLASS_H
```

15.7 learningCpp/Basics/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 }

15.7.1 Function Documentation

15.7.1.1 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
```

```

    // integral characters
    char char_a = ’c’; // always 1 byte
    wchar_t char_b = ’c’; // at least 1 byte
    //char8_t char_c = ’c’; // C++20
    //char16_t char_d = ’c’; // C++11 // at least 2 bytes
    //char32_t char_e = ’c’; // C++11 // at least 4 bytes

    // 0b12 --> binary
    // 012 --> octal
    // 0x12 --> hexadecimal
    // use std::dec , std::oct , std::hex

    // Integers
    short int_a = 1; // at least 2 bytes
    int int_b = 1; // at least 2 bytes
    long int_c = 1; // at least 4 bytes
    //long long int_d = 1; // C++11

    // Boolean
    bool bool_a = true; // or false

    // Null pointer
    //std::nullptr_t null_pointer = nullptr;

    // void

    // using cstdint
    //std::int8_t
    //std::uint8_t
    //std::int16_t
    //std::uint16_t
    //std::int32_t
    //std::uint32_t
    //std::int64_t
    //std::uint64_t

    // there is also the std::int_fast#_t providing the fastest signed integer with at least # bits
    // there is also the std::int_least#_t providing the smallest signed integer with at least # bits

    std::cout « "bool:\t\t" « sizeof(bool) « " bytes\n";
    std::cout « "char:\t\t" « sizeof(char) « " bytes\n";
    std::cout « "wchar_t:\t" « sizeof(wchar_t) « " bytes\n";
    std::cout « "char16_t:\t" « sizeof(char16_t) « " bytes\n"; // C++11 only
    std::cout « "char32_t:\t" « sizeof(char32_t) « " bytes\n"; // C++11 only
    std::cout « "short:\t\t" « sizeof(short) « " bytes\n";
    std::cout « "int:\t\t" « sizeof(int) « " bytes\n";
    std::cout « "long:\t\t" « sizeof(long) « " bytes\n";
    std::cout « "long long:\t" « sizeof(long long) « " bytes\n"; // C++11 only
    std::cout « "float:\t\t" « sizeof(float) « " bytes\n";
    std::cout « "double:\t\t" « sizeof(double) « " bytes\n";
    std::cout « "long double:\t" « sizeof(long double) « " bytes\n";

    // use const
    //const int const_int = 1;
    // for variables that should not be modifiable after initialization
    // and whose initializer is NOT known at compile-time

    // use constexpr
    //constexpr int constexpr_int = 1;
    // for variables that should not be modifiable after initialization
    // and whose initializer is known at compile-time
    for (int i = 0; i < 5; i++) {
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        std::cout « "\a"; // makes an alert
    }
    std::cout « "Backspace \b" « std::endl;
    std::cout « "Formfeed \f" « std::endl;
    std::cout « "Newline \n" « std::endl;
    std::cout « "Carriage return \r" « std::endl;
    std::cout « "Horizontal \t tab" « std::endl;
    std::cout « "Vertical tab \v" « std::endl;
    std::cout « "Single quote \’ or double quote \"" « std::endl;
    std::cout « "Octal number \12" « std::endl;
    std::cout « "Hex number \x14" « std::endl;
    int x_1 = 2;
    int x_2 = 3;
    int max_x = (x_1 > x_2) ? x_1 : x_2;
    // define a namespace
    //namespace namespace_1 {
    //    //nested namespace
    //    namespace namespace_1_nested {

    //    }
    //}
    // accessible using "::"

    // namespace alias
    // namespace nested_namespace = namespace_1::namespace_1_nested;
    // static local variables are not destroyed when out of scope (in contrast to automatic)
    static int var_1 { 1 };
    // AVOID using static variables unless the variable never needs to be reset
    typedef double distance_t; // define distance_t as an alias for type double
    //which is equivalent to: using distance_t = double;
    // The following two statements are equivalent:
    // double howFar; //equivalent to
    distance_t howFar;
    // IMPLICIT type conversion (coercion)
    float f_int { 3 }; // initializing floating point variable with int 3

    // EXPLICIT type conversion
    // static_cast
    int i1 { 10 };
    int i2 { 4 };
    // convert an int to a float so we get floating point division rather than integer division
    float f { static_cast<float>(i1) / i2 };

    enum Color
    {
        color_black, // assigned 0
        color_red, // assigned 1
        color_blue, // assigned 2
        color_green, // assigned 3
        color_white, // assigned 4
        color_cyan, // assigned 5
        color_yellow, // assigned 6
        color_magenta // assigned 7
    };
    Color paint{ color_white };
    std::cout « paint;

    // enum classes (scoped enumerations)
    enum class Fruit
    {
        banana, // banana is inside the scope of Fruit
        apple
    };
    Fruit fruit{ Fruit::banana }; // note: banana is not directly accessible any more, we have to use Fruit::banana
    struct Employee
    {
        short id;
        int age;
        double wage;
    };

    Employee joe{ 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
    Employee frank{ 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default initialization)

    //Employee joe; // create an Employee struct for Joe
    //joe.id = 14; // assign a value to member id within struct joe
    //joe.age = 32; // assign a value to member age within struct joe
    //joe.wage = 24.15; // assign a value to member wage within struct joe

    //Employee frank; // create an Employee struct for Frank
    //frank.id = 15; // assign a value to member id within struct frank
    //frank.age = 28; // assign a value to member age within struct frank
    //frank.wage = 18.27; // assign a value to member wage within struct frank

    // nested structs
    struct Company
    {
        Employee CEO; // Employee is a struct within the Company struct
        int numberOfEmployees;
    };
    Company myCompany{{ 1, 42, 60000.0 }, 5 };
    // halt (using <cstdlib>)
    //std::exit(0); // terminate and return 0 to operating system
    // ATTENTION: be aware of leaking resources

    // Conditional branches
    if (true) {

    } else if (false) {

    } else {

    }
    // init statements
    //    if (std::string fullName{ firstName + ’ ’ + lastName }; fullName.length() > 20)
    //    {
    //        std::cout « ’"’ « fullName « "\"is too long!\n";
    //    }
```

```
//    else
//    {
//        std::cout « "Your name is " « fullName « '\n';
//    }

// Switch statements
Color color {color_black};
switch (color)
{
    case Color::color_black:
        std::cout « "Black";
        break;
    case Color::color_white:
        std::cout « "White";
        break;
    case Color::color_red:
        std::cout « "Red";
        break;
        //[[fallthrough]];
    case Color::color_green:
        std::cout « "Green";
        break;
    case Color::color_blue:
        std::cout « "Blue";
        break;
    default:
        std::cout « "Unknown";
        break;
}
//[[fallthrough]] attribute can be added to indicate that the fall-through is intentional.

// Goto statements
//tryAgain:
//    goto tryAgain;

// While statements
int while_counter{ 5 };
while (while_counter < 10) {
    std::cout « "while_counter: " « while_counter « std::endl;
    ++while_counter;
}

// Do wile statements
do {
    std::cout « "while_counter: " « while_counter « std::endl;
    ++while_counter;
}
while (while_counter < 15);

// For statements
for (int count{ 0 }; count < 10; ++count)
    std::cout « count « ' ';
int iii{};
int jjj{};
for (iii = 0, jjj = 9; iii < 10; ++iii, --jjj)
    std::cout « iii « ' ' « jjj « '\n';
// return statement terminates the entire function the loop is within
// break terminates the loop
// continue jumps to the end of the loop body for the current iteration
//int prime[5]{}; // hold the first 5 prime numbers
//prime[0] = 2; // The first element has index 0
//prime[1] = 3;
//prime[2] = 5;
//prime[3] = 7;
//prime[4] = 11; // The last element has index 4 (array length-1)
int prime[5]{ 2, 3, 5, 7, 11 }; // use initializer list to initialize the fixed array
//int prime[]{ 2, 3, 5, 7, 11 }; // works as well
//std::cout « "The array has: " « std::size(prime) « " elements\n"; // C++17
//sizeof() gives the array length multiplied by element size

// Multidimensional arrays
int num_rows{3};
int num_cols{5};
int multi_dim_array[3][5] // cannot use num_rows or num_cols --> see dynamic memory allocation
        {
                { 1, 2, 3, 4, 5 }, // row 0
                { 6, 7, 8, 9, 10 }, // row 1
                { 11, 12, 13, 14, 15 } // row 2
        };
for (int row{ 0 }; row < num_rows; ++row) // step through the rows in the array
{
    for (int col{ 0 }; col < num_cols; ++col) // step through each element in the row
    {
        std::cout « multi_dim_array[row][col];
    }
}

```

```
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 }
```

15.7.2 Variable Documentation

15.7.2.1 g_global_integer

`int g_global_integer { 1 }`

Global variables

Definition at line 13 of file Basics.cpp.

15.7.2.2 g_x_1

`int g_x_1  [static]`

Definition at line 18 of file Basics.cpp.

15.7.2.3 g_x_2

`const int g_x_2 { 2 }  [extern]`

15.8 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
```

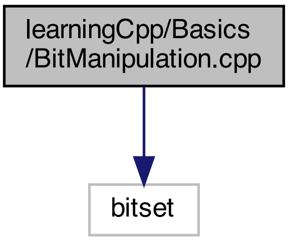
```
// integral characters
char char_a = ’c’; // always 1 byte
wchar_t char_b = ’c’; // at least 1 byte
//char8_t char_c = ’c’; // C++20
//char16_t char_d = ’c’; // C++11 // at least 2 bytes
//char32_t char_e = ’c’; // C++11 // at least 4 bytes

// 0b12 --> binary
// 012 --> octal
// 0x12 --> hexadecimal
// use std::dec , std::oct , std::hex

// Integers
short int_a = 1; // at least 2 bytes
int int_b = 1; // at least 2 bytes
long int_c = 1; // at least 4 bytes
//long long int_d = 1; // C++11

// Boolean
bool bool_a = true; // or false

// Null pointer
//std::nullptr_t null_pointer = nullptr;

// void

// using cstdint
//std::int8_t
//std::uint8_t
//std::int16_t
//std::uint16_t
//std::int32_t
//std::uint32_t
//std::int64_t
//std::uint64_t

// there is also the std::int_fast#_t providing the fastest signed integer with at least # bits
// there is also the std::int_least#_t providing the smallest signed integer with at least # bits

std::cout « "bool:\t\t" « sizeof(bool) « " bytes\n";
std::cout « "char:\t\t" « sizeof(char) « " bytes\n";
std::cout « "wchar_t:\t" « sizeof(wchar_t) « " bytes\n";
std::cout « "char16_t:\t" « sizeof(char16_t) « " bytes\n"; // C++11 only
std::cout « "char32_t:\t" « sizeof(char32_t) « " bytes\n"; // C++11 only
std::cout « "short:\t\t" « sizeof(short) « " bytes\n";
std::cout « "int:\t\t" « sizeof(int) « " bytes\n";
std::cout « "long:\t\t" « sizeof(long) « " bytes\n";
std::cout « "long long:\t" « sizeof(long long) « " bytes\n"; // C++11 only
std::cout « "float:\t\t" « sizeof(float) « " bytes\n";
std::cout « "double:\t\t" « sizeof(double) « " bytes\n";
std::cout « "long double:\t" « sizeof(long double) « " bytes\n";

// use const
//const int const_int = 1;
// for variables that should not be modifiable after initialization
// and whose initializer is NOT known at compile-time

// use constexpr
//constexpr int constexpr_int = 1;
// for variables that should not be modifiable after initialization
// and whose initializer is known at compile-time
for (int i = 0; i < 5; i++) {
    std::this_thread::sleep_for(std::chrono::milliseconds(250));
    std::cout « "\a"; // makes an alert
}
std::cout « "Backspace \b" « std::endl;
std::cout « "Formfeed \f" « std::endl;
std::cout « "Newline \n" « std::endl;
std::cout « "Carriage return \r" « std::endl;
std::cout « "Horizontal \t tab" « std::endl;
std::cout « "Vertical tab \v" « std::endl;
std::cout « "Single quote \’ or double quote \"" « std::endl;
std::cout « "Octal number \12" « std::endl;
std::cout « "Hex number \x14" « std::endl;
int x_1 = 2;
int x_2 = 3;
int max_x = (x_1 > x_2) ? x_1 : x_2;
// define a namespace
//namespace namespace_1 {
//    //nested namespace
//    namespace namespace_1_nested {

//    }
//}
// accessible using "::"

// namespace alias
```

```
    // namespace nested_namespace = namespace_1::namespace_1_nested;
    // static local variables are not destroyed when out of scope (in contrast to automatic)
    static int var_1 { 1 };
    // AVOID using static variables unless the variable never needs to be reset
    typedef double distance_t; // define distance_t as an alias for type double
    //which is equivalent to: using distance_t = double;
    // The following two statements are equivalent:
    // double howFar; //equivalent to
    distance_t howFar;
    // IMPLICIT type conversion (coercion)
    float f_int { 3 }; // initializing floating point variable with int 3

    // EXPLICIT type conversion
    // static_cast
    int i1 { 10 };
    int i2 { 4 };
    // convert an int to a float so we get floating point division rather than integer division
    float f { static_cast<float>(i1) / i2 };

    enum Color
    {
        color_black, // assigned 0
        color_red, // assigned 1
        color_blue, // assigned 2
        color_green, // assigned 3
        color_white, // assigned 4
        color_cyan, // assigned 5
        color_yellow, // assigned 6
        color_magenta // assigned 7
    };
    Color paint{ color_white };
    std::cout « paint;

    // enum classes (scoped enumerations)
    enum class Fruit
    {
        banana, // banana is inside the scope of Fruit
        apple
    };
    Fruit fruit{ Fruit::banana }; // note: banana is not directly accessible any more, we have to use Fruit::banana
    struct Employee
    {
        short id;
        int age;
        double wage;
    };

    Employee joe{ 1, 32, 60000.0 }; // joe.id = 1, joe.age = 32, joe.wage = 60000.0
    Employee frank{ 2, 28 }; // frank.id = 2, frank.age = 28, frank.wage = 0.0 (default initialization)

    //Employee joe; // create an Employee struct for Joe
    //joe.id = 14; // assign a value to member id within struct joe
    //joe.age = 32; // assign a value to member age within struct joe
    //joe.wage = 24.15; // assign a value to member wage within struct joe

    //Employee frank; // create an Employee struct for Frank
    //frank.id = 15; // assign a value to member id within struct frank
    //frank.age = 28; // assign a value to member age within struct frank
    //frank.wage = 18.27; // assign a value to member wage within struct frank

    // nested structs
    struct Company
    {
        Employee CEO; // Employee is a struct within the Company struct
        int numberOfEmployees;
    };
    Company myCompany{{ 1, 42, 60000.0 }, 5 };
    // halt (using <cstdlib>)
    //std::exit(0); // terminate and return 0 to operating system
    // ATTENTION: be aware of leaking resources

    // Conditional branches
    if (true) {

    } else if (false) {

    } else {

    }
    // init statements
    //    if (std::string fullName{ firstName + ’ ’ + lastName }; fullName.length() > 20)
    //    {
    //        std::cout « ’"’ « fullName « "\"is too long!\n";
    //    }
    //    else
    //    {
    //        std::cout « "Your name is " « fullName « '\n';
    //    }

    // Switch statements
    Color color {color_black};
    switch (color)
    {
        case Color::color_black:
            std::cout « "Black";
            break;
        case Color::color_white:
            std::cout « "White";
            break;
        case Color::color_red:
            std::cout « "Red";
            break;
            //[[fallthrough]];
        case Color::color_green:
            std::cout « "Green";
            break;
        case Color::color_blue:
            std::cout « "Blue";
            break;
        default:
            std::cout « "Unknown";
            break;
    }
    //[[fallthrough]] attribute can be added to indicate that the fall-through is intentional.

    // Goto statements
    //tryAgain:
    //    goto tryAgain;

    // While statements
    int while_counter{ 5 };
    while (while_counter < 10) {
        std::cout « "while_counter: " « while_counter « std::endl;
        ++while_counter;
    }

    // Do wile statements
    do {
        std::cout « "while_counter: " « while_counter « std::endl;
        ++while_counter;
    }
    while (while_counter < 15);

    // For statements
    for (int count{ 0 }; count < 10; ++count)
        std::cout « count « ' ';
    int iii{};
    int jjj{};
    for (iii = 0, jjj = 9; iii < 10; ++iii, --jjj)
        std::cout « iii « ' ' « jjj « '\n';
    // return statement terminates the entire function the loop is within
    // break terminates the loop
    // continue jumps to the end of the loop body for the current iteration
    //int prime[5]{}; // hold the first 5 prime numbers
    //prime[0] = 2; // The first element has index 0
    //prime[1] = 3;
    //prime[2] = 5;
    //prime[3] = 7;
    //prime[4] = 11; // The last element has index 4 (array length-1)
    int prime[5]{ 2, 3, 5, 7, 11 }; // use initializer list to initialize the fixed array
    //int prime[]{ 2, 3, 5, 7, 11 }; // works as well
    //std::cout « "The array has: " « std::size(prime) « " elements\n"; // C++17
    //sizeof() gives the array length multiplied by element size

    // Multidimensional arrays
    int num_rows{3};
    int num_cols{5};
    int multi_dim_array[3][5] // cannot use num_rows or num_cols --> see dynamic memory allocation
            {
                    { 1, 2, 3, 4, 5 }, // row 0
                    { 6, 7, 8, 9, 10 }, // row 1
                    { 11, 12, 13, 14, 15 } // row 2
            };
    for (int row{ 0 }; row < num_rows; ++row) // step through the rows in the array
    {
        for (int col{ 0 }; col < num_cols; ++col) // step through each element in the row
        {
            std::cout « multi_dim_array[row][col];
        }
    }

    // foreach loop
    for (auto &element: prime)
    {
        std::cout «  element « std::endl;
    }
    return 0; // 0, EXIT_SUCCESS, EXIT_FAILURE
}
```

15.9 learningCpp/Basics/BitManipulation.cpp File Reference

`#include <bitset>`

Include dependency graph for BitManipulation.cpp:



Functions

- int main ()

15.9.1 Function Documentation

15.9.1.1 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
```

```
    constexpr std::uint_fast8_t mask3{ 0b0000’1000 }; // represents bit 3
    constexpr std::uint_fast8_t mask4{ 0b0001’0000 }; // represents bit 4
    constexpr std::uint_fast8_t mask5{ 0b0010’0000 }; // represents bit 5
    constexpr std::uint_fast8_t mask6{ 0b0100’0000 }; // represents bit 6
    constexpr std::uint_fast8_t mask7{ 0b1000’0000 }; // represents bit 7
    // C++11 or earlier
    //    constexpr std::uint_fast8_t mask0{ 0x1 }; // hex for 0000 0001
    //    constexpr std::uint_fast8_t mask1{ 0x2 }; // hex for 0000 0010
    //    constexpr std::uint_fast8_t mask2{ 0x4 }; // hex for 0000 0100
    //    constexpr std::uint_fast8_t mask3{ 0x8 }; // hex for 0000 1000
    //    constexpr std::uint_fast8_t mask4{ 0x10 }; // hex for 0001 0000
    //    constexpr std::uint_fast8_t mask5{ 0x20 }; // hex for 0010 0000
    //    constexpr std::uint_fast8_t mask6{ 0x40 }; // hex for 0100 0000
    //    constexpr std::uint_fast8_t mask7{ 0x80 }; // hex for 1000 0000
    //    // or
    //    constexpr std::uint_fast8_t mask0{ 1 « 0 }; // 0000 0001
    //    constexpr std::uint_fast8_t mask1{ 1 « 1 }; // 0000 0010
    //    constexpr std::uint_fast8_t mask2{ 1 « 2 }; // 0000 0100
    //    constexpr std::uint_fast8_t mask3{ 1 « 3 }; // 0000 1000
    //    constexpr std::uint_fast8_t mask4{ 1 « 4 }; // 0001 0000
    //    constexpr std::uint_fast8_t mask5{ 1 « 5 }; // 0010 0000
    //    constexpr std::uint_fast8_t mask6{ 1 « 6 }; // 0100 0000
    //    constexpr std::uint_fast8_t mask7{ 1 « 7 }; // 1000 0000
    return 0;
}
```

15.10 BitManipulation.cpp

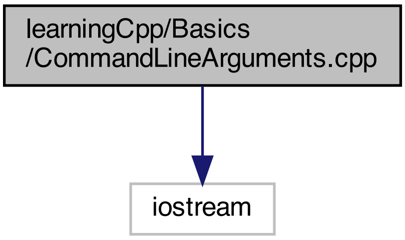
```
//
// Created by Michael Staneker on 01.12.20.
//

#include <bitset>

int main() {

    std::bitset<8> bits{ 0b0000’0101 }; // we need 8 bits, start with bit pattern 0000 0101
    bits.set(3); // set bit position 3 to 1 (now we have 0000 1101)
    bits.flip(4); // flip bit 4 (now we have 0001 1101)
    bits.reset(4); // set bit 4 back to 0 (now we have 0000 1101)

    std::cout « "All the bits: " « bits « ’\n’;
    std::cout « "Bit 3 has value: " « bits.test(3) « ’\n’;
    std::cout « "Bit 4 has value: " « bits.test(4) « ’\n’;

    // x « y // left shift
    // x » y // right shift
    // ~x // bitwise NOT
    // x & y // bitwise AND
    // x | y // bitwise OR
    // x ^ y // bitwise XOR
    // x «= < // left shift assignment
    // x »= y // right shift assignment
    // x |= y // bitwise OR assignment
    // x &= y // bitwise AND assignment
    // x ^= y // bitwise XOR assignment
    // since C++14
    constexpr std::uint_fast8_t mask0{ 0b0000’0001 }; // represents bit 0
    constexpr std::uint_fast8_t mask1{ 0b0000’0010 }; // represents bit 1
    constexpr std::uint_fast8_t mask2{ 0b0000’0100 }; // represents bit 2
    constexpr std::uint_fast8_t mask3{ 0b0000’1000 }; // represents bit 3
    constexpr std::uint_fast8_t mask4{ 0b0001’0000 }; // represents bit 4
    constexpr std::uint_fast8_t mask5{ 0b0010’0000 }; // represents bit 5
    constexpr std::uint_fast8_t mask6{ 0b0100’0000 }; // represents bit 6
    constexpr std::uint_fast8_t mask7{ 0b1000’0000 }; // represents bit 7
    // C++11 or earlier
    //    constexpr std::uint_fast8_t mask0{ 0x1 }; // hex for 0000 0001
    //    constexpr std::uint_fast8_t mask1{ 0x2 }; // hex for 0000 0010
    //    constexpr std::uint_fast8_t mask2{ 0x4 }; // hex for 0000 0100
    //    constexpr std::uint_fast8_t mask3{ 0x8 }; // hex for 0000 1000
    //    constexpr std::uint_fast8_t mask4{ 0x10 }; // hex for 0001 0000
    //    constexpr std::uint_fast8_t mask5{ 0x20 }; // hex for 0010 0000
    //    constexpr std::uint_fast8_t mask6{ 0x40 }; // hex for 0100 0000
    //    constexpr std::uint_fast8_t mask7{ 0x80 }; // hex for 1000 0000
    //    // or
    //    constexpr std::uint_fast8_t mask0{ 1 « 0 }; // 0000 0001
    //    constexpr std::uint_fast8_t mask1{ 1 « 1 }; // 0000 0010
    //    constexpr std::uint_fast8_t mask2{ 1 « 2 }; // 0000 0100
    //    constexpr std::uint_fast8_t mask3{ 1 « 3 }; // 0000 1000
    //    constexpr std::uint_fast8_t mask4{ 1 « 4 }; // 0001 0000
    //    constexpr std::uint_fast8_t mask5{ 1 « 5 }; // 0010 0000
    //    constexpr std::uint_fast8_t mask6{ 1 « 6 }; // 0100 0000
    //    constexpr std::uint_fast8_t mask7{ 1 « 7 }; // 1000 0000
    return 0;
}
```

```
00064
```

15.11 learningCpp/Basics/CommandLineArguments.cpp File Reference

`#include <iostream>`

Include dependency graph for CommandLineArguments.cpp:



Functions

- int main (int argc, char ∗argv[ ])

15.11.1 Function Documentation

15.11.1.1 main()

```
int main (
            int argc,
            char ∗ argv[ ] )
```

15.11.2 Command line arguments

In order to pass command line arguments to the program use

- `int main(int argc, char *argv[ ])`
- `int main(int argc, char** argv)` both are treated identically!
- **argc** is an integer parameter containing a count of the number of arguments passed to the program, whereas **argc** is always at least 1, since the first argument is always the name of the program itself
- **argv** is where the actual arguments are stored (within an array of C-style strings)

Definition at line 21 of file CommandLineArguments.cpp.

```
00022 {
00023     std::cout « "There are " « argc « " arguments:\n";
00024
00025     // Loop through each argument and print its number and value
00026     for (int count{ 0 }; count < argc; ++count)
00027     {
00028         std::cout « count « ' ' « argv[count] « '\n';
00029     }
00030
00031     // handle numeric values
00032     //std::stringstream convert{ argv[1] }; // set up a stringstream variable named convert,
      initialized with the input from argv[1]
00033     //int myint{};
```

```
    //if (!(convert » myint)) // do the conversion
    //    myint = 0; // if conversion fails, set myint to a default value
    //std::cout « "Got integer: " « myint « '\n';

    return 0;
}
```

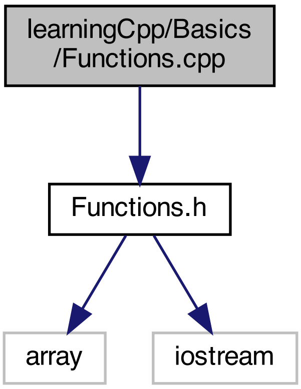
15.12 CommandLineArguments.cpp

```
//
// Created by Michael Staneker on 09.12.20.
//

#include <iostream>

int main(int argc, char *argv[])
{
    std::cout « "There are " « argc « " arguments:\n";

    // Loop through each argument and print its number and value
    for (int count{ 0 }; count < argc; ++count)
    {
        std::cout « count « ' ' « argv[count] « '\n';
    }

    // handle numeric values
    //std::stringstream convert{ argv[1] }; // set up a stringstream variable named convert, initialized with the input from argv[1]
    //int myint{};
    //if (!(convert » myint)) // do the conversion
    //    myint = 0; // if conversion fails, set myint to a default value
    //std::cout « "Got integer: " « myint « '\n';

    return 0;
}
```

15.13 learningCpp/Basics/Functions.cpp File Reference

`#include "Functions.h"`

Include dependency graph for Functions.cpp:



Functions

- void pass_by_value (int x)

  Function passing argument by value.

- void pass_by_reference (int &x)

  Function passing argument by reference.
- void pass_by_address (int ∗ptr)

  Function passing argument by address.
- int return_by_value ()

  Function returning value by value.
- int & return_by_reference ()

  Function returning value by reference.
- int ∗ return_by_address ()

  Function returning value by address.
- int overload_add (int a, int b)

  Function adding two values.
- int overload_add (int a, int b, int c)

  (Overloaded) Function adding three values
- void func_default_arg (int x, int y)

  Function with default argument (optional parameter)
- void countDown (int count)

  A simple recursive function.
- void lambda_example (std::array< std::string_view, 4 > arr)

  A simple lambda function.
- void ellipsis_example (int count,...)

  A simple function using ellipsis.
- template<typename T >
  T max (T x, T y)

  A simple template function.
- int main ()

15.13.1 Function Documentation

15.13.1.1 countDown()

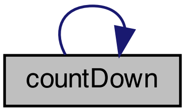
```
void countDown (
              int count )
```

A simple recursive function.

Definition at line 57 of file Functions.cpp.

```
00058 {
00059     std::cout « "push " « count « '\n';
00060
00061     if (count > 1) // termination condition
00062         countDown(count-1);
00063
00064     std::cout « "pop " « count « '\n';
00065 }
```

Here is the call graph for this function:



15.13.1.2 ellipsis_example()

```
void ellipsis_example (
              int count,
               ... )
```

A simple function using ellipsis.

Definition at line 83 of file Functions.cpp.

```
00083                                       {
00084     double sum{ 0 };
00085 
00086     // We access the ellipsis through a va_list, so let’s declare one
00087     va_list list;
00088 
00089     // We initialize the va_list using va_start.  The first parameter is
00090     // the list to initialize.  The second parameter is the last non-ellipsis
00091     // parameter.
00092     va_start(list, count);
00093 
00094     // Loop through all the ellipsis arguments
00095     for (int arg{ 0 }; arg < count; ++arg)
00096     {
00097         // We use va_arg to get parameters out of our ellipsis
00098         // The first parameter is the va_list we’re using
00099         // The second parameter is the type of the parameter
00100         sum += va_arg(list, int);
00101     }
00102 
00103     // Cleanup the va_list when we’re done.
00104     va_end(list);
00105 
00106     std::cout « "average = " « sum / count « std::endl;
00107 }
```

15.13.1.3 func_default_arg()

```
void func_default_arg (
              int x,
              int y )
```

Function with default argument (optional parameter)

Definition at line 52 of file Functions.cpp.

```
00052                                     {
00053     std::cout « "x = " « x « std::endl;
00054     std::cout « "y = " « y « std::endl;
00055 }
```

15.13.1.4 lambda_example()

```
void lambda_example (
              std::array< std::string_view, 4 > arr )
```

A simple lambda function.

Definition at line 67 of file Functions.cpp.

```
00067                                                      {
00068     const auto found{ std::find_if(arr.begin(), arr.end(),
00069                                    [](std::string_view str) // here’s our lambda, no capture clause
00070                                    {
00071                                        return (str.find("nut") != std::string_view::npos);
00072                                    }) };
00073 
00074     if (found == arr.end())
00075     {
00076         std::cout « "No nuts\n";
00077     }
00078     else {
00079         std::cout « "Found " « *found « ’\n’;
00080     }
00081 }
```

15.13.1.5 main()

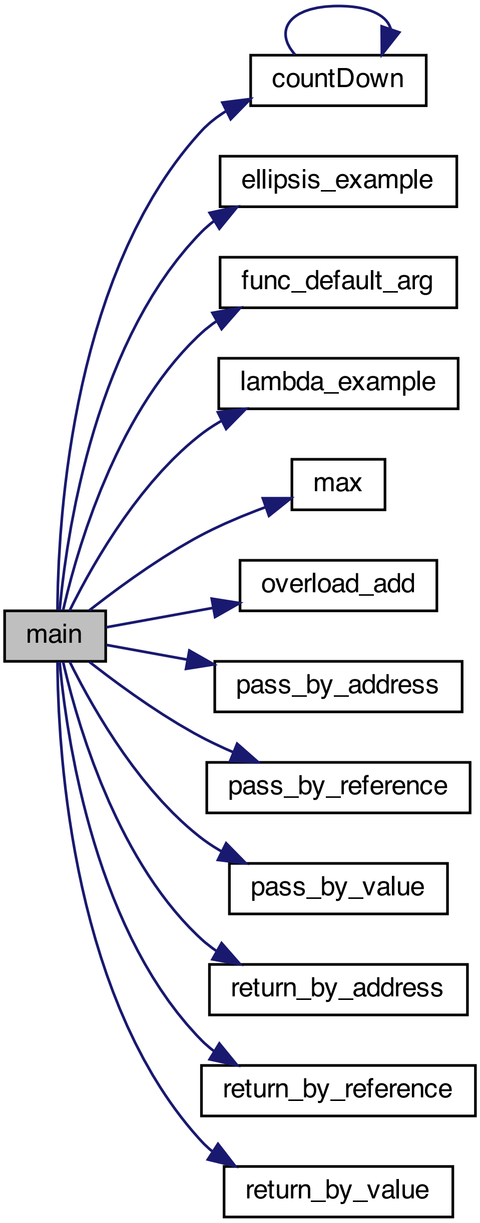
```
int main ( )
```

calling a function through a function pointer calling a recursive function

Definition at line 114 of file Functions.cpp.

```
00114            {
00115 
00116     int x = 5;
00117     pass_by_value(x);
00118 
00119     pass_by_reference(x);
00120     std::cout « "after passed by reference: x = " « x « std::endl;
00121 
00122     pass_by_address(&x);
00123     std::cout « "after passed by address x = " « x « std::endl;
00124 
00125     int value = return_by_value();
00126     std::cout « "return by value: value = " « value « std::endl;
00127 
00128     int *value_ptr = return_by_address();
00129     std::cout « "return by address: " « std::endl;
00130     std::cout « " value_ptr = " « value_ptr « std::endl;
00131     std::cout « "*value_ptr = " « *value_ptr « std::endl;
00132 
00133     int value_ref = return_by_reference();
00134     std::cout « "return by reference: value = " « value_ref « std::endl;
00135 
00136     int a = 1;
00137     int b = 2;
00138     int c = 3;
00139 
00140     int result_1 = overload_add(a, b);
00141     std::cout « "result_1 = " « result_1 « std::endl;
00142     int result_2 = overload_add(a, b, c);
00143     std::cout « "result_2 = " « result_2 « std::endl;
00144 
00145     std::cout « "func_default_arg(int x, int y=10) with x = 2" « std::endl;
00146     func_default_arg(2);
00147     std::cout « "func_default_arg(int x, int y=10) with x = 2 and y = 5" « std::endl;
00148     func_default_arg(2, 5);
00149 
00154     void (*fcnPtr)(int, int){ &func_default_arg }; // Initialize fcnPtr
00155     std::cout « "call function through function pointer: " « std::endl;
00156     fcnPtr(5, 5); // call function
00157 
00161     std::cout « "Calling a recursive function" « std::endl;
00162     countDown(5);
00163 
00164     std::cout « "ellipsis_example(2, 1, 5)" « std::endl;
00165     ellipsis_example(2, 1, 5);
00166 
00167     std::cout « "ellipsis_example(4, 1, 5, 7, 10)" « std::endl;
00168     ellipsis_example(4, 1, 5, 7, 10);
00169 
00170 
00171     std::cout « "lambda_example()" « std::endl;
00172     std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
00173     lambda_example(arr);
00174 
00175     int int_1 = 1;
00176     int int_2 = 2;
00177     int int_max = max(int_1, int_2);
00178     std::cout « "max integer = " « int_max « std::endl;
00179 
00180     double double_1 = 4.7;
00181     double double_2 = 7.9;
00182     double double_max = max(double_1, double_2);
00183     std::cout « "max double = " « double_max « std::endl;
00184 
00185     return 0;
00186 }
```

Here is the call graph for this function:



15.13.1.6 max()

```
template<typename T >
T max (
            T x,
            T y )
```

A simple template function.

Definition at line 109 of file Functions.cpp.

```
00109                                       {
00110     return (x > y) ? x : y;
00111 }
```

15.13.1.7 overload_add() [1/2]

```
int overload_add (
             int a,
             int b )
```

Function adding two values.

Definition at line 42 of file Functions.cpp.

```
00042                                {
00043     std::cout « "overload_add(int a, int b)" « std::endl;
00044     return a + b;
00045 }
```

15.13.1.8 overload_add() [2/2]

```
int overload_add (
             int a,
             int b,
             int c )
```

(Overloaded) Function adding three values

Definition at line 47 of file Functions.cpp.

```
00047                                       {
00048     std::cout « "overload_add(int a, int b, int c)" « std::endl;
00049     return a + b + c;
00050 }
```

15.13.1.9 pass_by_address()

```
void pass_by_address (
             int * ptr )
```

Function passing argument by address.

Definition at line 18 of file Functions.cpp.

```
00018                                {
00019     std::cout « "func: pass_by_address(int *ptr)" « std::endl;
00020     std::cout « "ptr = 4" « std::endl;
00021     *ptr = 4;
00022 }
```

15.13.1.10 pass_by_reference()

```
void pass_by_reference (
             int & x )
```

Function passing argument by reference.

Definition at line 12 of file Functions.cpp.

```
00012                                {
00013     std::cout « "func: pass_by_reference(int &x)" « std::endl;
00014     std::cout « "x += 1" « std::endl;
00015     x = x + 1;
00016 }
```

15.13.1.11 pass_by_value()

```
void pass_by_value (
             int x )
```

Function passing argument by value.

Definition at line 7 of file Functions.cpp.

```
00007                           {
00008     std::cout « "func: pass_by_value(int x)" « std::endl;
00009     std::cout « "x = " « x « std::endl;
00010 }
```

15.13.1.12 return_by_address()

`int* return_by_address ( )`

Function returning value by address.

Definition at line 36 of file Functions.cpp.

```
00036                         {
00037     std::cout « "func: return_by_address()" « std::endl;
00038     int value{ 2 };
00039     return &value; // return value by address
00040 } // value destroyed here
```

15.13.1.13 return_by_reference()

`int& return_by_reference ( )`

Function returning value by reference.

Definition at line 30 of file Functions.cpp.

```
00030                           {
00031     std::cout « "func: return_by_reference()" « std::endl;
00032     int value{ 2 };
00033     return value; // return a refernce to value
00034 }
```

15.13.1.14 return_by_value()

`int return_by_value ( )`

Function returning value by value.

Definition at line 24 of file Functions.cpp.

```
00024                       {
00025     std::cout « "func: return_by_value()" « std::endl;
00026     int value{ 2 };
00027     return value; // a copy of value will be returned
00028 } // value desroyed here
```

15.14 Functions.cpp

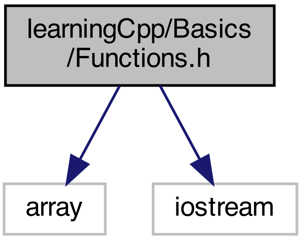
```
00001 //
00002 // Created by Michael Staneker on 08.12.20.
00003 //
00004
00005 #include "Functions.h"
00006
00007 void pass_by_value(int x) {
00008     std::cout « "func: pass_by_value(int x)" « std::endl;
00009     std::cout « "x = " « x « std::endl;
00010 }
00011
00012 void pass_by_reference(int &x) {
00013     std::cout « "func: pass_by_reference(int &x)" « std::endl;
00014     std::cout « "x += 1" « std::endl;
00015     x = x + 1;
00016 }
00017
00018 void pass_by_address(int *ptr) {
00019     std::cout « "func: pass_by_address(int *ptr)" « std::endl;
00020     std::cout « "ptr = 4" « std::endl;
00021     *ptr = 4;
00022 }
00023
00024 int return_by_value() {
00025     std::cout « "func: return_by_value()" « std::endl;
00026     int value{ 2 };
00027     return value; // a copy of value will be returned
00028 } // value desroyed here
00029
00030 int& return_by_reference() {
00031     std::cout « "func: return_by_reference()" « std::endl;
00032     int value{ 2 };
00033     return value; // return a refernce to value
00034 }
00035
```

```
int* return_by_address() {
    std::cout « "func: return_by_address()" « std::endl;
    int value{ 2 };
    return &value; // return value by address
} // value destroyed here

int overload_add(int a, int b) {
    std::cout « "overload_add(int a, int b)" « std::endl;
    return a + b;
}

int overload_add(int a, int b, int c) {
    std::cout « "overload_add(int a, int b, int c)" « std::endl;
    return a + b + c;
}

void func_default_arg(int x, int y) {
    std::cout « "x = " « x « std::endl;
    std::cout « "y = " « y « std::endl;
}

void countDown(int count)
{
    std::cout « "push " « count « '\n';

    if (count > 1) // termination condition
        countDown(count-1);

    std::cout « "pop " « count « '\n';
}

void lambda_example(std::array<std::string_view, 4> arr) {
    const auto found{ std::find_if(arr.begin(), arr.end(),
                                   [](std::string_view str) // here's our lambda, no capture clause
                                   {
                                       return (str.find("nut") != std::string_view::npos);
                                   }) };

    if (found == arr.end())
    {
        std::cout « "No nuts\n";
    }
    else {
        std::cout « "Found " « *found « '\n';
    }
}

void ellipsis_example(int count, ...) {
    double sum{ 0 };

    // We access the ellipsis through a va_list, so let's declare one
    va_list list;

    // We initialize the va_list using va_start.  The first parameter is
    // the list to initialize.  The second parameter is the last non-ellipsis
    // parameter.
    va_start(list, count);

    // Loop through all the ellipsis arguments
    for (int arg{ 0 }; arg < count; ++arg)
    {
        // We use va_arg to get parameters out of our ellipsis
        // The first parameter is the va_list we're using
        // The second parameter is the type of the parameter
        sum += va_arg(list, int);
    }

    // Cleanup the va_list when we're done.
    va_end(list);

    std::cout « "average = " « sum / count « std::endl;
}

template <typename T> T max(T x, T y) {
    return (x > y) ? x : y;
}


int main() {

    int x = 5;
    pass_by_value(x);

    pass_by_reference(x);
    std::cout « "after passed by reference: x = " « x « std::endl;

    pass_by_address(&x);
    std::cout « "after passed by address x = " « x « std::endl;

    int value = return_by_value();
    std::cout « "return by value: value = " « value « std::endl;

    int *value_ptr = return_by_address();
    std::cout « "return by address: " « std::endl;
    std::cout « " value_ptr = " « value_ptr « std::endl;
    std::cout « "*value_ptr = " « *value_ptr « std::endl;

    int value_ref = return_by_reference();
    std::cout « "return by reference: value = " « value_ref « std::endl;

    int a = 1;
    int b = 2;
    int c = 3;

    int result_1 = overload_add(a, b);
    std::cout « "result_1 = " « result_1 « std::endl;
    int result_2 = overload_add(a, b, c);
    std::cout « "result_2 = " « result_2 « std::endl;

    std::cout « "func_default_arg(int x, int y=10) with x = 2" « std::endl;
    func_default_arg(2);
    std::cout « "func_default_arg(int x, int y=10) with x = 2 and y = 5" « std::endl;
    func_default_arg(2, 5);

    void (*fcnPtr)(int, int){ &func_default_arg }; // Initialize fcnPtr
    std::cout « "call function through function pointer: " « std::endl;
    fcnPtr(5, 5); // call function

    std::cout « "Calling a recursive function" « std::endl;
    countDown(5);

    std::cout « "ellipsis_example(2, 1, 5)" « std::endl;
    ellipsis_example(2, 1, 5);

    std::cout « "ellipsis_example(4, 1, 5, 7, 10)" « std::endl;
    ellipsis_example(4, 1, 5, 7, 10);


    std::cout « "lambda_example()" « std::endl;
    std::array<std::string_view, 4> arr{ "apple", "banana", "walnut", "lemon" };
    lambda_example(arr);

    int int_1 = 1;
    int int_2 = 2;
    int int_max = max(int_1, int_2);
    std::cout « "max integer = " « int_max « std::endl;

    double double_1 = 4.7;
    double double_2 = 7.9;
    double double_max = max(double_1, double_2);
    std::cout « "max double = " « double_max « std::endl;

    return 0;
}
```

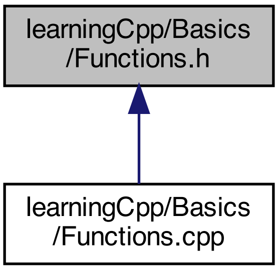
15.15 learningCpp/Basics/Functions.h File Reference

```
#include <array>
#include <iostream>
```

Include dependency graph for Functions.h:



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



Functions

- void pass_by_value (int x)

    Function passing argument by value.
- void pass_by_reference (int &x)

    Function passing argument by reference.
- void pass_by_address (int ∗x)

    Function passing argument by address.
- int return_by_value ()

    Function returning value by value.
- int & return_by_reference ()

    Function returning value by reference.
- int ∗ return_by_address ()

    Function returning value by address.
- int overload_add (int a, int b)

    Function adding two values.
- int overload_add (int a, int b, int c)

    (Overloaded) Function adding three values
- void func_default_arg (int x, int y=10)

Function with default argument (optional parameter)

- void countDown (int count)

  A simple recursive function.

- void ellipsis_example (int count,...)

  A simple function using ellipsis.

- void lambda_example (std::array< std::string_view, 4 > arr)

  A simple lambda function.

- template<typename T >
  T max (T x, T y)

  A simple template function.

15.15.1 Detailed Description

15.15.2 Function parameters and arguments

15.15.2.1 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!

15.15.2.1.1 Pros

- Arguments can be anything
- Arguments are never changed by the function (prevents possibly unwanted side effects)

15.15.2.1.2 Cons

- Copying classes and structs can incur a significant performance penalty

15.15.2.1.3 When to use

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

15.15.2.1.4 When not to use

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

15.15.2.2 Pass by reference

To **pass** a variable **by reference**, simply declare the function parameters as references. When the function is called the function parameter becomes a reference to the argument being called with, so that any changes made to the reference are passed through to the argument!
Since functions can have only one return value, references or rather reference function parameters can be used to return multiple values.

15.15.2.2.1 read-only pass by reference If it is undesirable to change an argument, make it read-only by passing it as **const reference**, so that an error occurs if the function tries to change the argument.
That's useful, since

- It enlists the compilers help in ensuring values that shouldn’t be changed aren’t changed (the compiler will throw an error if you try, like in the above example).
- It tells the programmer that the function won’t change the value of the argument. This can help with debugging.
- You can’t pass a const argument to a non-const reference parameter. Using const parameters ensures you can pass both non-const and const arguments to the function.
- Const references can accept any type of argument, including non-const l-values, const l-values, and r-values.

15.15.2.2.2 Pros

- References allow a function to change the value of the argument, which is sometimes useful. Otherwise, const references can be used to guarantee the function won't change the argument.

- Because a copy of the argument is not made, pass by reference is fast, even when used with large structs or classes.

- References can be used to return multiple values from a function (via out parameters).

- References must be initialized, so there's no worry about null values.

15.15.2.2.3 Cons

- Because a non-const reference cannot be initialized with a const l-value or an r-value (e.g. a literal or an expression), arguments to non-const reference parameters must be normal variables.

- It can be hard to tell whether an argument passed by non-const reference is meant to be input, output, or both. Judicious use of const and a naming suffix for out variables can help.

- It's impossible to tell from the function call whether the argument may change. An argument passed by value and passed by reference looks the same. We can only tell whether an argument is passed by value or reference by looking at the function declaration. This can lead to situations where the programmer does not realize a function will change the value of the argument.

15.15.2.2.4 When to use

- When passing structs or classes (const if read-only is wanted)

- When the function needs to modify an argument

- When the type information of a fixed array is required

15.15.2.3 Pass by address

There is one more way to pass variables to functions, and that is by address. **Passing an argument by address** involves passing the address of the argument variable rather than the argument variable itself. Because the argument is an address, the function parameter must be a pointer. The function can then dereference the pointer to access or change the value being pointed to.
Remember that fixed arrays decay into pointers when passed to a function, therefore the length of the array has to be passed separately.
Addresses are actually passed by value, so by changing the function parameter itself, the original pointer argument will not be changed.
To change the address an argument points to, just pass the addresses by reference.

15.15.2.3.1 read-only pass by address If it is undesirable to change an argument, make it read-only by passing it as **const pointer**, so that an error occurs if the function tries to change the argument.

15.15.2.3.2 Pros

- Pass by address allows a function to change the value of the argument, which is sometimes useful. Otherwise, const can be used to guarantee the function won't change the argument. (However, if you want to do this with a non-pointer, you should use pass by reference instead).

- Because a copy of the argument is not made, it is fast, even when used with large structs or classes.

- We can return multiple values from a function via out parameters.

15.15.2.3.3 Cons

- Because literals (excepting C-style string literals) and expressions do not have addresses, pointer arguments must be normal variables.
- All values must be checked to see whether they are null. Trying to dereference a null value will result in a crash. It is easy to forget to do this.
- Because dereferencing a pointer is slower than accessing a value directly, accessing arguments passed by address is slower than accessing arguments passed by value.

15.15.2.3.4 When to use

- When passing built-in arrays (if you're okay with the fact that they'll decay into a pointer).
- When passing a pointer and nullptr is a valid argument logically.

15.15.2.3.5 When not to use

- When passing a pointer and nullptr is not a valid argument logically (use pass by reference).
- When passing structs or classes (use pass by reference).
- When passing fundamental types (use pass by value).

15.15.2.4 Returning values by value, reference and address

This is quite similar to passing arguments to functions. In fact, returning values from a function to its caller by value, address or reference **works almost exactly the same way as passing arguments to a function, with the same upsides and downsides**.
Attention: Since local variables in a function go out of scope and are destroyed, this needs to be considered!

15.15.2.4.1 Return by value To use, when

- returning variables that were declared inside the function
- returning function arguments that were passed by value

Not to use, when

- returning a built-in array or pointer (use return by address instead!)
- returning a large struct or class (use return by reference instead!)

15.15.2.4.2 Return by address Not possible for literals or expressions (no address)!
To use, when

- returning dynamically allocated memory and there is no type to handle the allocation
- returning function arguments that were passed by address

Not to use, when

- returning variables declared inside the function or parameters passed by value (use return by value instead!)
- returning a large struct or class that was passed by reference (use return by refernce instead!)

15.15.2.5 Return by reference

Not possible for literals or expressions (no address)!
To use, when

- returning a reference parameter
- returning a member of an object that was passed into the function by reference or address
- returning a large struct or class that will not be destroyed at the end of the function

Not to use, when

- returning variables declared inside the function or parameters passed by value (use return by value instead!)
- returning a built-in array or pointer value (use return by address)

15.15.2.6 Returning multiple values

To return multiple values

- a struct (or in principle a class)
- std::tuple can be used!

15.15.3 Function overloading

Function overloading is a feature of C++ that allows us to create multiple functions with the same name, so long as they have different parameters.
Attention: Function return types are not considered distinct!

15.15.4 Default arguments

A default argument is a default value provided for a function parameter. If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used. If the user does supply an argument for the parameter, the user-supplied argument is used. Because the user can choose whether to supply a specific argument value, or use the default, a parameter with a default value provided is often called an optional parameter.

15.15.5 Function pointers

Function pointers point to functions!

- non-const function pointer: int (_fcnPtr)()_
- _const function pointer: int (∗const fcnPtr)()_
- _use type aliases to make function pointers ∗prettier_: using ValidateFunction = bool(∗)(int, int)
- using std::function (from e.g. <functional>): std::function<bool(int, int)> fcn

Using function pointers, functions can be passed to other functions as arguments.

15.15.6 The stack and the heap

The memory a program uses is divided into:

- The code segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
- The bss segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
- The data segment (also called the initialized data segment), where initialized global and static variables are stored.
- The heap, where dynamically allocated variables are allocated from.
- The call stack, where function parameters, local variables, and other function-related information are stored.

15.15.6.1 Heap

The heap segment (also known as the “free store”) keeps track of memory used for dynamic memory allocation (when using *new* and *delete*).

15.15.6.2 Stack

The call stack (usually referred to as “the stack”) has a much more interesting role to play. The call stack keeps track of all the active functions (those that have been called but have not yet terminated) from the start of the program to the current point of execution, and handles allocation of all function parameters and local variables.

15.15.7 Recursion

Recursive functions are functions that call themselves.

15.15.8 Variable number of parameters - Ellipsis

All parameters a function will take must be known in advance (even if they have default values). However, there are certain cases where it can be useful to be able to pass a variable number of parameters to a function. C++ provides a special specifier known as ellipsis ∗...∗.
Avoid using ellipsis, for many reasons ... !

15.15.9 Lambdas - anonymous functions

See `Lambdas` for reference!
A **lambda expression** (also *lambda* or *closure*) allows to define an anonymous function inside another function.
The syntax is:

```
 {C++}
[ captureClause ] ( parameters ) -> returnType
{
    statements;
}
```

Consequently a trivial lambda looks like: `[]() {}`.
In actuality, lambdas aren’t functions (which is part of how they avoid the limitation of C++ not supporting nested functions). They’re a special kind of object called a **functor**. **Functors** are objects that contain an overloaded operator() that make them callable like a function.
Use auto when initializing variables with lambdas, and std::function if you can’t initialize the variable with the lambda.

15.15.10 Function templates

Definition in file Functions.h.

15.15.11 Function Documentation

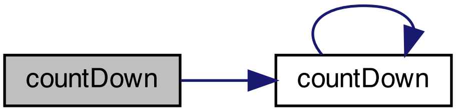
15.15.11.1 countDown()

```
void countDown (
               int count )
```

A simple recursive function.
Definition at line 57 of file Functions.cpp.

```
00058 {
00059     std::cout « "push " « count « '\n';
00060 
00061     if (count > 1) // termination condition
00062         countDown(count-1);
00063 
00064     std::cout « "pop " « count « '\n';
00065 }
```

Here is the call graph for this function:



15.15.11.2 ellipsis_example()

```
void ellipsis_example (
             int count,
              ... )
```

A simple function using ellipsis.

Definition at line 83 of file Functions.cpp.

```
00083                                     {
00084     double sum{ 0 };
00085 
00086     // We access the ellipsis through a va_list, so let's declare one
00087     va_list list;
00088 
00089     // We initialize the va_list using va_start.  The first parameter is
00090     // the list to initialize.  The second parameter is the last non-ellipsis
00091     // parameter.
00092     va_start(list, count);
00093 
00094     // Loop through all the ellipsis arguments
00095     for (int arg{ 0 }; arg < count; ++arg)
00096     {
00097         // We use va_arg to get parameters out of our ellipsis
00098         // The first parameter is the va_list we're using
00099         // The second parameter is the type of the parameter
00100         sum += va_arg(list, int);
00101     }
00102 
00103     // Cleanup the va_list when we're done.
00104     va_end(list);
00105 
00106     std::cout « "average = " « sum / count « std::endl;
00107 }
```

15.15.11.3 func_default_arg()

```
void func_default_arg (
             int x,
             int y = 10 )
```

Function with default argument (optional parameter)

Definition at line 52 of file Functions.cpp.

```
00052                                  {
00053     std::cout « "x = " « x « std::endl;
00054     std::cout « "y = " « y « std::endl;
00055 }
```

15.15.11.4 lambda_example()

```
void lambda_example (
             std::array< std::string_view, 4 > arr )
```

A simple lambda function.

Definition at line 67 of file Functions.cpp.

```
00067                                                     {
00068     const auto found{ std::find_if(arr.begin(), arr.end(),
00069                                   [](std::string_view str) // here’s our lambda, no capture clause
00070                                   {
00071                                       return (str.find("nut") != std::string_view::npos);
00072                                   }) };
00073 
00074     if (found == arr.end())
00075     {
00076         std::cout « "No nuts\n";
00077     }
00078     else {
00079         std::cout « "Found " « *found « ’\n’;
00080     }
00081 }
```

15.15.11.5 max()

```
template<typename T >
T max (
              T x,
              T y )
```

A simple template function.

Definition at line 109 of file Functions.cpp.

```
00109                                    {
00110     return (x > y) ? x : y;
00111 }
```

15.15.11.6 overload_add() [1/2]

```
int overload_add (
              int a,
              int b )
```

Function adding two values.

Definition at line 42 of file Functions.cpp.

```
00042                              {
00043     std::cout « "overload_add(int a, int b)" « std::endl;
00044     return a + b;
00045 }
```

15.15.11.7 overload_add() [2/2]

```
int overload_add (
              int a,
              int b,
              int c )
```

(Overloaded) Function adding three values

Definition at line 47 of file Functions.cpp.

```
00047                                    {
00048     std::cout « "overload_add(int a, int b, int c)" « std::endl;
00049     return a + b + c;
00050 }
```

15.15.11.8 pass_by_address()

```
void pass_by_address (
              int * x )
```

Function passing argument by address.

Definition at line 18 of file Functions.cpp.

```
00018                              {
00019     std::cout « "func: pass_by_address(int *ptr)" « std::endl;
00020     std::cout « "ptr = 4" « std::endl;
00021     *ptr = 4;
00022 }
```

15.15.11.9 pass_by_reference()

```
void pass_by_reference (
              int & x )
```

Function passing argument by reference.

Definition at line 12 of file Functions.cpp.

```
00012                                    {
00013     std::cout « "func: pass_by_reference(int &x)" « std::endl;
00014     std::cout « "x += 1" « std::endl;
00015     x = x + 1;
00016 }
```

15.15.11.10 pass_by_value()

```
void pass_by_value (
              int x )
```

Function passing argument by value.

Definition at line 7 of file Functions.cpp.

```
00007                               {
00008     std::cout « "func: pass_by_value(int x)" « std::endl;
00009     std::cout « "x = " « x « std::endl;
00010 }
```

15.15.11.11 return_by_address()

```
int* return_by_address ( )
```

Function returning value by address.

Definition at line 36 of file Functions.cpp.

```
00036                           {
00037     std::cout « "func: return_by_address()" « std::endl;
00038     int value{ 2 };
00039     return &value; // return value by address
00040 } // value destroyed here
```

15.15.11.12 return_by_reference()

```
int& return_by_reference ( )
```

Function returning value by reference.

Definition at line 30 of file Functions.cpp.

```
00030                             {
00031     std::cout « "func: return_by_reference()" « std::endl;
00032     int value{ 2 };
00033     return value; // return a refernce to value
00034 }
```

15.15.11.13 return_by_value()

```
int return_by_value ( )
```

Function returning value by value.

Definition at line 24 of file Functions.cpp.

```
00024                         {
00025     std::cout « "func: return_by_value()" « std::endl;
00026     int value{ 2 };
00027     return value; // a copy of value will be returned
00028 } // value desroyed here
```

15.16 Functions.h

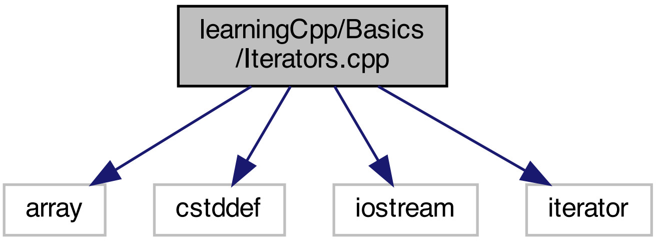
```
00001 //
00002 // Created by Michael Staneker on 08.12.20.
00003 //
00004 
00005 #include <array>
00006 #include <iostream>
00007 
```

```
#ifndef CPP_TEMPLATE_PROJECT_FUNCTIONS_H
#define CPP_TEMPLATE_PROJECT_FUNCTIONS_H

// \fn void pass_by_value(int)

void pass_by_value(int x);

void pass_by_reference(int &x);

void pass_by_address(int *x);


int return_by_value();

int& return_by_reference();

int* return_by_address();

int overload_add(int a, int b);

int overload_add(int a, int b, int c);


void func_default_arg(int x, int y=10);

void countDown(int count);

void ellipsis_example(int count, ...);

void lambda_example(std::array<std::string_view, 4> arr);

template <typename T> T max(T x, T y);




#endif //CPP_TEMPLATE_PROJECT_FUNCTIONS_H
```

15.17 learningCpp/Basics/Iterators.cpp File Reference

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

Include dependency graph for Iterators.cpp:



Functions

- int main ()

15.17.1 Function Documentation

15.17.1.1 main()

```
int main ( )
```

Iterators

Definition at line 6 of file Iterators.cpp.

```
{
    // The type is automatically deduced to std::array<int, 7> (Requires C++17).
    // Use the type std::array<int, 7> if your compiler doesn’t support C++17.
    std::array<int, 7> data{ 0, 1, 2, 3, 4, 5, 6 };
    std::size_t length{ std::size(data) };

    // while-loop with explicit index
    std::cout « "While loop with explicit index" « std::endl;
    std::size_t index{ 0 };
    while (index != length)
    {
        std::cout « data[index] « ’ ’;
        ++index;
    }
    std::cout « ’\n’;

    // for-loop with explicit index
    std::cout « "For loop with explicit index" « std::endl;
    for (index = 0; index < length; ++index)
    {
        std::cout « data[index] « ’ ’;
    }
    std::cout « ’\n’;

    // for-loop with pointer (Note: ptr can’t be const, because we increment it)
    std::cout « "For loop with pointer" « std::endl;
    for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
    {
        std::cout « *ptr « ’ ’;
    }
    std::cout « ’\n’;

    // ranged-based for loop
    std::cout « "Range based for loop" « std::endl;
    for (int i : data)
    {
        std::cout « i « ’ ’;
    }
    std::cout « ’\n’;

    std::cout « std::endl;

    // Pointers (simplest kind of Iterators)
    std::cout « "Iterator: Pointer..." « std::endl;
    auto begin{ &data[0] };
    // note that this points to one spot beyond the last element
    auto end{ begin + std::size(data) };

    // for-loop with pointer
    for (auto ptr{ begin }; ptr != end; ++ptr) // ++ to move to next element
    {
        std::cout « *ptr « ’ ’; // Indirection to get value of current element
    }
    std::cout « ’\n’;

    // Standard library iterators
    std::cout « "Standard library iterators..." « std::endl;
    // Ask our array for the begin and end points (via the begin and end member functions).
    begin = { data.begin() };
    end = { data.end() };

    for (auto p{ begin }; p != end; ++p) // ++ to move to next element.
    {
        std::cout « *p « ’ ’; // Indirection to get value of current element.
    }
    std::cout « ’\n’;

    //
    std::cout « "or..." « std::endl;

    begin = { std::begin(data) };
    end = { std::end(data) };

    for (auto p{ begin }; p != end; ++p) // ++ to move to next element
    {
        std::cout « *p « ’ ’; // Indirection to get value of current element
    }
    std::cout « ’\n’;

    return 0;
}
```

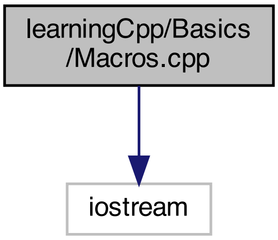
15.18 Iterators.cpp

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

int main()
{
    // The type is automatically deduced to std::array<int, 7> (Requires C++17).
    // Use the type std::array<int, 7> if your compiler doesn’t support C++17.
    std::array<int, 7> data{ 0, 1, 2, 3, 4, 5, 6 };
    std::size_t length{ std::size(data) };

    // while-loop with explicit index
    std::cout « "While loop with explicit index" « std::endl;
    std::size_t index{ 0 };
    while (index != length)
    {
        std::cout « data[index] « ’ ’;
        ++index;
    }
    std::cout « ’\n’;

    // for-loop with explicit index
    std::cout « "For loop with explicit index" « std::endl;
    for (index = 0; index < length; ++index)
    {
        std::cout « data[index] « ’ ’;
    }
    std::cout « ’\n’;

    // for-loop with pointer (Note: ptr can’t be const, because we increment it)
    std::cout « "For loop with pointer" « std::endl;
    for (auto ptr{ &data[0] }; ptr != (&data[0] + length); ++ptr)
    {
        std::cout « *ptr « ’ ’;
    }
    std::cout « ’\n’;

    // ranged-based for loop
    std::cout « "Range based for loop" « std::endl;
    for (int i : data)
    {
        std::cout « i « ’ ’;
    }
    std::cout « ’\n’;

    std::cout « std::endl;

    // Pointers (simplest kind of Iterators)
    std::cout « "Iterator: Pointer..." « std::endl;
    auto begin{ &data[0] };
    // note that this points to one spot beyond the last element
    auto end{ begin + std::size(data) };

    // for-loop with pointer
    for (auto ptr{ begin }; ptr != end; ++ptr) // ++ to move to next element
    {
        std::cout « *ptr « ’ ’; // Indirection to get value of current element
    }
    std::cout « ’\n’;

    // Standard library iterators
    std::cout « "Standard library iterators..." « std::endl;
    // Ask our array for the begin and end points (via the begin and end member functions).
    begin = { data.begin() };
    end = { data.end() };

    for (auto p{ begin }; p != end; ++p) // ++ to move to next element.
    {
        std::cout « *p « ’ ’; // Indirection to get value of current element.
    }
    std::cout « ’\n’;

    //
    std::cout « "or..." « std::endl;

    begin = { std::begin(data) };
    end = { std::end(data) };

    for (auto p{ begin }; p != end; ++p) // ++ to move to next element
    {
        std::cout « *p « ’ ’; // Indirection to get value of current element
    }
    std::cout « ’\n’;

    return 0;
}
```

15.19 learningCpp/Basics/Macros.cpp File Reference

`#include <iostream>`

Include dependency graph for Macros.cpp:



Macros

- #define PI 3.1415
- #define EULER

Functions

- int main ()

15.19.1 Macro Definition Documentation

15.19.1.1 EULER

`#define EULER`

Definition at line 21 of file Macros.cpp.

15.19.1.2 PI

`#define PI 3.1415`

Header guards (conditional compilation directive)

Definition at line 18 of file Macros.cpp.

15.19.2 Function Documentation

15.19.2.1 main()

`int main ( )`

Definition at line 25 of file Macros.cpp.

```
00025           {
00026
#ifdef PI // or if not defined use #ifndef
    std::cout « "PI is: " « PI « std::endl
//#elif
//#else
#endif

#ifdef EULER
    std::cout « "EULER is defined, but not replaceable, or rather replaceable by empty"
#endif

    return 0;
}
```

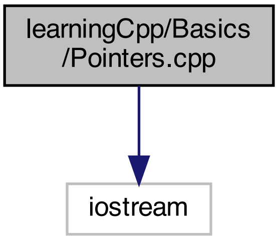
15.20 Macros.cpp

```
//
// Created by Michael Staneker on 01.12.20.
//

#include <iostream>

//#ifnedf SOME_UNIQUE_NAME_HERE
//#define SOME_UNIQUE_NAME_HERE
//
//#endif

// or alternatively use, but bit supported by all compilers
//#pragma once
// define macro (with substitution text)
#define PI 3.1415

// empty substitution text
#define EULER



int main() {

#ifdef PI // or if not defined use #ifndef
    std::cout « "PI is: " « PI « std::endl
//#elif
//#else
#endif

#ifdef EULER
    std::cout « "EULER is defined, but not replaceable, or rather replaceable by empty"
#endif

    return 0;
}
```

15.21 learningCpp/Basics/Pointers.cpp File Reference

`#include <iostream>`

Include dependency graph for Pointers.cpp:



Functions

- int main ()
  Brief description.

15.21.1 Function Documentation

15.21.1.1 main()

```
int main ( )
```

Brief description.

15.21.2 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

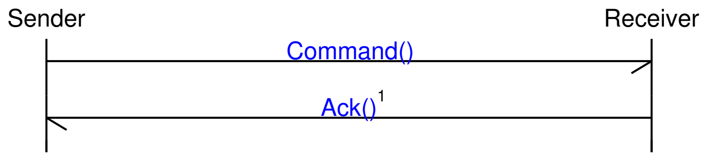
Parameters

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

15.21.2.1 Address operator &

15.21.2.2 Indirection operator $*$

15.21.2.3 Pointers

15.21.2.3.1 pointer

15.21.2.3.2 Pointers

15.21.2.3.3 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)
```

15.21.2.3.4 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;
```

15.21.2.3.5 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
```

15.21.2.3.6 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 };
```

```
    std::cout « *ptr_for_array « ’\n’; // will print 9

    // ARRAYS DECAY INTO POINTERS WHEN PASSED TO FUNCTIONS !!!
    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)
    //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;
    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
    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
    return 0;

}
```

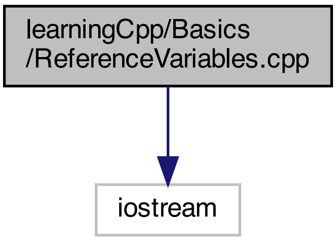
15.22 Pointers.cpp

```
//
// Created by Michael Staneker on 03.12.20.
//

#include <iostream>

int main() {

    int x{ 5 };
    std::cout « "    x = " « x « ’\n’; // print the value of variable x

    std::cout « "   &x = " « &x « ’\n’; // print the memory address of variable x
    std::cout « "*(&x) = " « *(&x) « ’\n’; // print the memory address of variable x
    //int *iPtr{}; // a pointer to an integer value
    //double *dPtr{}; // a pointer to a double value
    //int* iPtr2{}; // also valid syntax (acceptable, but not favored)
    //int * iPtr3{}; // also valid syntax (but don’t do this, it looks like multiplication)
    //int *iPtr4{}, *iPtr5{}; // declare two pointers to integer variables (not recommended)

    int var{ 5 };
    int *ptr{ &var }; // initialize ptr with address of variable v
    std::cout « "var = " « var « ’\n’; // print the address of variable v
    std::cout « "var = " « &var « ’\n’; // print the address of variable v
    std::cout « "ptr = " « ptr « ’\n’; // print the address that ptr is holding
    std::cout « "*ptr = " « *ptr « ’\n’;

    //    Pointers are good for:
    //    * dynamic arrays
    //    * dynamically allocate memory
    //    * pass large amount of data to a function (without copying)
    //    * pass a function as a parameter to another function
    //    * achieve polymorphism when dealing with inheritance
    //    * useful for advanced data structures
    //assigning it to the literal 0
    float *null_ptr { 0 };  // ptr is now a null pointer
    float *null_ptr2; // ptr2 is uninitialized
    null_ptr2 = 0; // ptr2 is now a null pointer
    float *null_ptr3 {nullptr}; // C++11
    int array[5]{ 9, 7, 5, 3, 1 };
    std::cout « *array « '\n'; // will print 9
    int *ptr_for_array{ array };
    std::cout « *ptr_for_array « '\n'; // will print 9

    // ARRAYS DECAY INTO POINTERS WHEN PASSED TO FUNCTIONS !!!
    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)
    //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;
    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
    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
    return 0;

}
```

15.23 learningCpp/Basics/ReferenceVariables.cpp File Reference

```
#include <iostream>
```

Include dependency graph for ReferenceVariables.cpp:



Functions

- int main ()

15.23.1 Function Documentation

15.23.1.1 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 }
```

15.24 ReferenceVariables.cpp

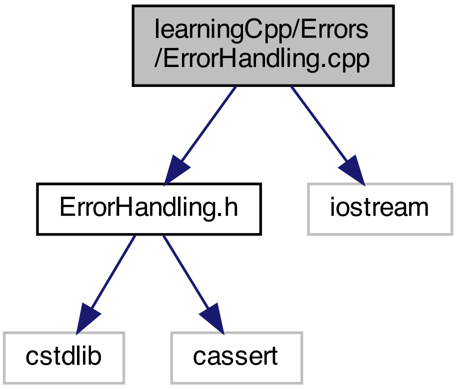
```
00001 //
00002 // Created by Michael Staneker on 03.12.20.
00003 //
00004 
00005 #include <iostream>
00006 
00007 int main() {
00008 
00009     int value {5};
00010 
00012     int &reference{ value }; // "reference to" value
```

```
    //int& reference{ value }; // valid
    //int & reference{ value }; // valid

    int x{ 5 }; // normal integer
    int &y{ x }; // y is a reference to x
    int &z{ y }; // z is also a reference to x

    std::cout « " x = " «  x « std::endl;
    std::cout « " y = " «  y « std::endl;
    std::cout « " z = " «  z « std::endl;
    std::cout « "&x = " « &x « std::endl;
    std::cout « "&y = " « &y « std::endl;
    std::cout « "&z = " « &z « std::endl;

    // References cannot be reassigned !
    // reference = value; // not valid


    return 0;
}

```

15.25 learningCpp/Errors/ErrorHandling.cpp File Reference

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

Include dependency graph for ErrorHandling.cpp:



Functions

- int returning_error_code ()
- void write_error_message ()
- void exit_program ()
- void assert_example (int x)
- int main ()

15.25.1 Function Documentation

15.25.1.1 assert_example()

```
void assert_example (
            int x )
```

Definition at line 22 of file ErrorHandling.cpp.

```
00022                       {
00023     std::cout « "assert() example" « std::endl;
00024     std::cout « "x = " « x « std::endl;
00025     //terminates if assert evaluates to true
00026     assert(x > 0);
00027 }
```

15.25.1.2 exit_program()

```
void exit_program ( )
```

Definition at line 17 of file ErrorHandling.cpp.

```
00017                    {
00018     std::cout « "exiting with error number 2 to OS" « std::endl;
00019     std::exit(2);
00020 }
```

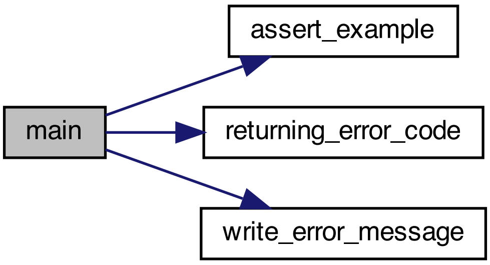
15.25.1.3 main()

```
int main ( )
```

Definition at line 29 of file ErrorHandling.cpp.

```
00029            {
00030 
00031     int error_code = returning_error_code();
00032     std::cout « "error code received from returning_error_code: " « error_code « std::endl;
00033 
00034     write_error_message();
00035 
00036     // assert() evaluates to false if x <= 0
00037     assert_example(1);
00038 
00039     //exit_program();
00040     //std::cout « "This shouldn't be printed!!!" « std::endl;
00041 
00042     return 0;
00043 
00044 }
```

Here is the call graph for this function:



15.25.1.4 returning_error_code()

```
int returning_error_code ( )
```

15.25.2 Error handling

Errors fall into two categories:

- **Syntax errors** occurring when a statement is not valid according to the grammar of the C++ language
- **Semantic errors** occurring when a statement is syntactically valid, but does not do what the programmer intended

15.25.2.1 Assert statements

An assert statement is a preprocessor macro that evaluates a conditional expression at runtime. If the conditional expression is true, the assert statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated. This error message contains the conditional expression that failed, along with the name of the code file and the line number of the assert.

15.25.2.1.1 assert() Include <cstdlib> for `assert()`, operating at runtime, which comes with a small performance cost that is incurred each time the assert condition is checked, which can be disabled by defining the **NDEBUG** macro.

15.25.2.1.2 static_assert() Another type of assert is `static_assert<condition, diagnostic_message>`, designed to operate at compile time.

Attention There is no chance for cleanup after terminating with exit() or assert()!

Definition at line 9 of file ErrorHandling.cpp.

```
00009                             {
00010     return -1;
00011 }
```

15.25.2.2 write_error_message()

`void write_error_message ( )`

Definition at line 13 of file ErrorHandling.cpp.

```
00013                             {
00014     std::cerr « "This is an error message!" « std::endl;
00015 }
```

15.26 ErrorHandling.cpp

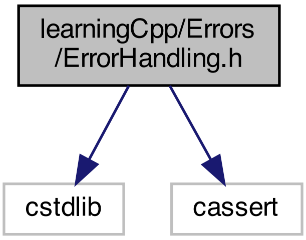
```
00001 //
00002 // Created by Michael Staneker on 09.12.20.
00003 //
00004 
00005 #include "ErrorHandling.h"
00006 
00007 #include <iostream>
00008 
00009 int returning_error_code() {
00010     return -1;
00011 }
00012 
00013 void write_error_message() {
00014     std::cerr « "This is an error message!" « std::endl;
00015 }
00016 
00017 void exit_program() {
00018     std::cout « "exiting with error number 2 to OS" « std::endl;
00019     std::exit(2);
00020 }
00021 
00022 void assert_example(int x) {
00023     std::cout « "assert() example" « std::endl;
00024     std::cout « "x = " « x « std::endl;
00025     //terminates if assert evaluates to true
00026     assert(x > 0);
00027 }
00028 
00029 int main() {
00030 
00031     int error_code = returning_error_code();
00032     std::cout « "error code received from returning_error_code: " « error_code « std::endl;
00033 
00034     write_error_message();
00035 
00036     // assert() evaluates to false if x <= 0
00037     assert_example(1);
```

```

    //exit_program();
    //std::cout « "This shouldn’t be printed!!!" « std::endl;

    return 0;

}
```

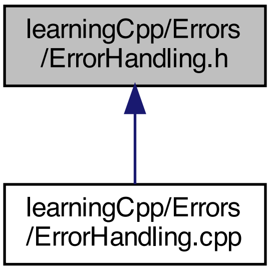
15.27 learningCpp/Errors/ErrorHandling.h File Reference

```
#include <cstdlib>
#include <cassert>
```

Include dependency graph for ErrorHandling.h:



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



Macros

- #define CPP_TEMPLATE_PROJECT_ERRORHANDLING_H

Functions

- int returning_error_code ()
- void write_error_message ()
- void exit_program ()
- void assert_example (int x)

15.27.1 Macro Definition Documentation

15.27.1.1 CPP_TEMPLATE_PROJECT_ERRORHANDLING_H

```
#define CPP_TEMPLATE_PROJECT_ERRORHANDLING_H
```

Definition at line 9 of file ErrorHandling.h.

15.27.2 Function Documentation

15.27.2.1 assert_example()

```
void assert_example (
             int x )
```

Definition at line 22 of file ErrorHandling.cpp.

```
00022                        {
00023     std::cout « "assert() example" « std::endl;
00024     std::cout « "x = " « x « std::endl;
00025     //terminates if assert evaluates to true
00026     assert(x > 0);
00027 }
```

15.27.2.2 exit_program()

```
void exit_program ( )
```

Definition at line 17 of file ErrorHandling.cpp.

```
00017                  {
00018     std::cout « "exiting with error number 2 to OS" « std::endl;
00019     std::exit(2);
00020 }
```

15.27.2.3 returning_error_code()

```
int returning_error_code ( )
```

15.27.3 Error handling

Errors fall into two categories:

- **Syntax errors** occurring when a statement is not valid according to the grammar of the C++ language
- **Semantic errors** occurring when a statement is syntactically valid, but does not do what the programmer intended

15.27.3.1 Assert statements

An assert statement is a preprocessor macro that evaluates a conditional expression at runtime. If the conditional expression is true, the assert statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated. This error message contains the conditional expression that failed, along with the name of the code file and the line number of the assert.

15.27.3.1.1 assert() Include <cstdlib> for `assert()`, operating at runtime, which comes with a small performance cost that is incurred each time the assert condition is checked, which can be disabled by defining the **NDEBUG** macro.

15.27.3.1.2 static_assert() Another type of assert is `static_assert<condition, diagnostic_message>`, designed to operate at compile time.

Attention There is no chance for cleanup after terminating with exit() or assert()!

Definition at line 9 of file ErrorHandling.cpp.

```
00009                         {
00010     return -1;
00011 }
```

15.27.3.2 write_error_message()

`void write_error_message ( )`

Definition at line 13 of file ErrorHandling.cpp.

```
00013                         {
00014     std::cerr « "This is an error message!" « std::endl;
00015 }
```

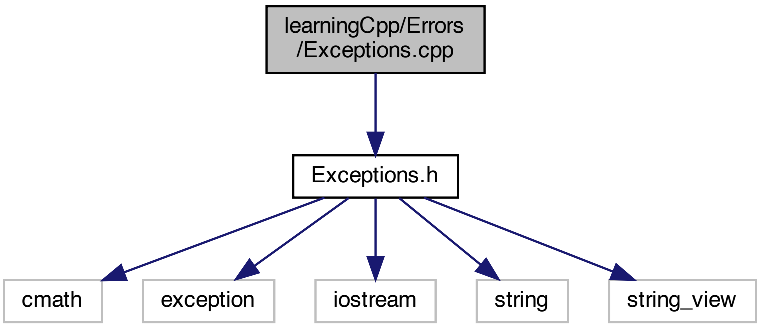
15.28 ErrorHandling.h

```
00001 //
00002 // Created by Michael Staneker on 09.12.20.
00003 //
00004
00005 #include <cstdlib> //for std::exit
00006 #include <cassert> //for assert()
00007
00008 #ifndef CPP_TEMPLATE_PROJECT_ERRORHANDLING_H
00009 #define CPP_TEMPLATE_PROJECT_ERRORHANDLING_H
00010
00040 int returning_error_code();
00041
00042 void write_error_message();
00043
00044 void exit_program();
00045
00046 void assert_example(int x);
00047
00048 #endif //CPP_TEMPLATE_PROJECT_ERRORHANDLING_H
```

15.29 learningCpp/Errors/Exceptions.cpp File Reference

`#include "Exceptions.h"`

Include dependency graph for Exceptions.cpp:



15.30 Exceptions.cpp

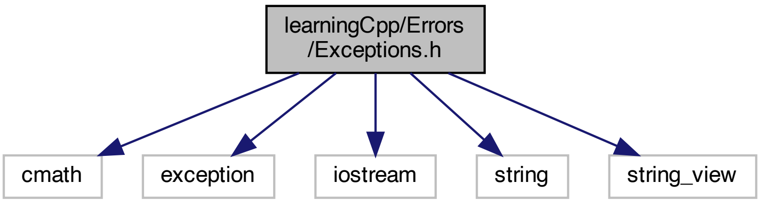
```
00001 //
00002 // Created by Michael Staneker on 10.12.20.
```

```
//

#include "Exceptions.h"


ArrayException::ArrayException(std::string_view error)
        : m_error{error}
{
}



int IntArray::getLength() const {
    return 3;
}

int& IntArray::operator[](const int index)
{
    if (index < 0 || index >= getLength())
        throw ArrayException("Invalid index");

    return m_data[index];
}


double SQRT::mySqrt(double x) {
    // If the user entered a negative number, this is an error condition
    if (x < 0.0)
        throw "Can not take sqrt of negative number"; // throw exception of type const char*

    return sqrt(x);
}
```

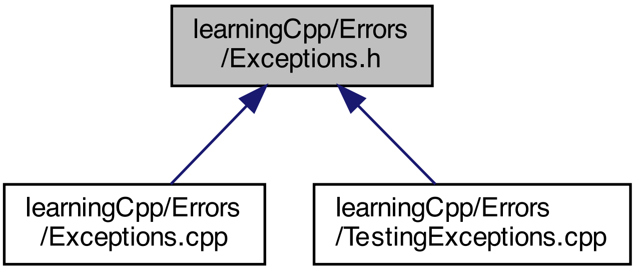
15.31 learningCpp/Errors/Exceptions.h File Reference

```
#include <cmath>
#include <exception>
#include <iostream>
#include <string>
#include <string_view>
```

Include dependency graph for Exceptions.h:



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



Classes

- class ArrayException
- class IntArray
- class SQRT
- class Exceptions

15.32 Exceptions.h

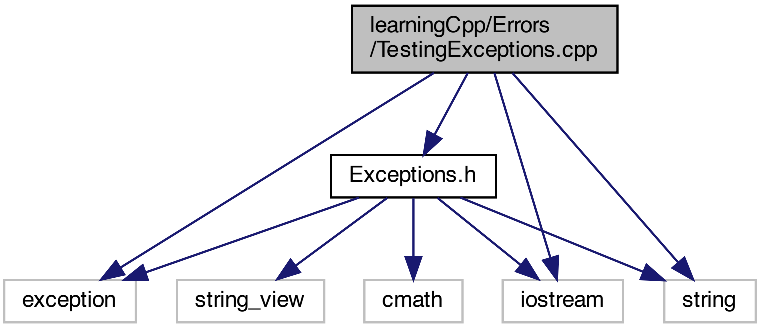
```
//
// Created by Michael Staneker on 10.12.20.
//
#ifndef CPP_TEMPLATE_PROJECT_EXCEPTIONS_H
#define CPP_TEMPLATE_PROJECT_EXCEPTIONS_H

#include <cmath>
#include <exception> // for std::exception
#include <iostream>
#include <string>
#include <string_view>

class ArrayException : public std::exception
{
private:
    std::string m_error{};

public:
    ArrayException(std::string_view error);
    // return the std::string as a const C-style string
    const char* what() const noexcept override {
        return m_error.c_str();
    }
};

class IntArray
{
private:

    int m_data[3]; // assume array is length 3 for simplicity
public:

    IntArray() {}

    int getLength() const;

    int& operator[](const int index);

};

class SQRT {
public:
    static double mySqrt(double x);
};
```

```
00137
00138
00139 class Exceptions {
00140
00141 };
00142
00143
00144 #endif //CPP_TEMPLATE_PROJECT_EXCEPTIONS_H
```

15.33 learningCpp/Errors/TestingExceptions.cpp File Reference

```
#include "Exceptions.h"
#include <iostream>
#include <string>
#include <exception>
```

Include dependency graph for TestingExceptions.cpp:



Functions

- int main ()

15.33.1 Function Documentation

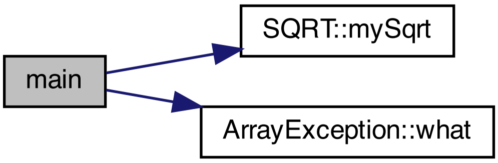
15.33.1.1 main()

```
int main ( )
```

Definition at line 11 of file TestingExceptions.cpp.

```
00011            {
00012
00013     try
00014     {
00015         // Statements that may throw exceptions you want to handle go here
00016         throw -1; // here's a trivial example
00017     }
00018     catch (int x)
00019     {
00020         // Any exceptions of type int thrown within the above try block get sent here
00021         std::cerr « "We caught an int exception with value: " « x « '\n';
00022     }
00023     catch (double) // no variable name since we don't use the exception itself in the catch block
       below
00024     {
00025         // Any exceptions of type double thrown within the above try block get sent here
00026         std::cerr « "We caught an exception of type double" « '\n';
00027     }
00028     catch (const std::string &str) // catch classes by const reference
00029     {
        // Any exceptions of type std::string thrown within the above try block get sent here
        std::cerr « "We caught an exception of type std::string" « ’\n’;
    }

    std::cout « "Continuing on our merry way\n";


    double x{ -1 };

    try // Look for exceptions that occur within try block and route to attached catch block(s)
    {
        double d = SQRT::mySqrt(x);
        std::cout « "The sqrt of " « x « " is " « d « ’\n’;
    }
    catch (const char* exception) // catch exceptions of type const char*
    {
        std::cerr « "Error: " « exception « std::endl;
    }

    try
    {
        // Your code using standard library goes here
        // We’ll trigger one of these exceptions intentionally for the sake of example
        std::string s;
        s.resize(-1); // will trigger a std::length_error
    }
        // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception &exception)
    {
        std::cerr « "Standard exception: " « exception.what() « ’\n’;
    }

    try
    {
        throw std::runtime_error("Bad things happened");
    }
        // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception &exception)
    {
        std::cerr « "Standard exception: " « exception.what() « ’\n’;
    }

    IntArray array;

    try
    {
        int value{ array[5] };
    }
    catch (const ArrayException &exception) // derived catch blocks go first
    {
        std::cerr « "An array exception occurred (" « exception.what() « ")\n";
    }
    catch (const std::exception &exception)
    {
        std::cerr « "Some other std::exception occurred (" « exception.what() « ")\n";
    }

    return 0;
}
```

Here is the call graph for this function:



15.34 TestingExceptions.cpp

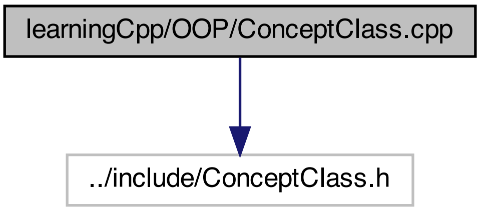
```
//
```

```
// Created by Michael Staneker on 10.12.20.
//

#include "Exceptions.h"

#include <iostream>
#include <string>
#include <exception>

int main() {

    try
    {
        // Statements that may throw exceptions you want to handle go here
        throw -1; // here’s a trivial example
    }
    catch (int x)
    {
        // Any exceptions of type int thrown within the above try block get sent here
        std::cerr « "We caught an int exception with value: " « x « ’\n’;
    }
    catch (double) // no variable name since we don’t use the exception itself in the catch block below
    {
        // Any exceptions of type double thrown within the above try block get sent here
        std::cerr « "We caught an exception of type double" « ’\n’;
    }
    catch (const std::string &str) // catch classes by const reference
    {
        // Any exceptions of type std::string thrown within the above try block get sent here
        std::cerr « "We caught an exception of type std::string" « ’\n’;
    }

    std::cout « "Continuing on our merry way\n";


    double x{ -1 };

    try // Look for exceptions that occur within try block and route to attached catch block(s)
    {
        double d = SQRT::mySqrt(x);
        std::cout « "The sqrt of " « x « " is " « d « ’\n’;
    }
    catch (const char* exception) // catch exceptions of type const char*
    {
        std::cerr « "Error: " « exception « std::endl;
    }

    try
    {
        // Your code using standard library goes here
        // We’ll trigger one of these exceptions intentionally for the sake of example
        std::string s;
        s.resize(-1); // will trigger a std::length_error
    }
        // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception &exception)
    {
        std::cerr « "Standard exception: " « exception.what() « ’\n’;
    }

    try
    {
        throw std::runtime_error("Bad things happened");
    }
        // This handler will catch std::exception and all the derived exceptions too
    catch (const std::exception &exception)
    {
        std::cerr « "Standard exception: " « exception.what() « ’\n’;
    }

    IntArray array;

    try
    {
        int value{ array[5] };
    }
    catch (const ArrayException &exception) // derived catch blocks go first
    {
        std::cerr « "An array exception occurred (" « exception.what() « ")\n";
    }
    catch (const std::exception &exception)
    {
        std::cerr « "Some other std::exception occurred (" « exception.what() « ")\n";
    }

    return 0;
}

```

15.35 learningCpp/OOP/ConceptClass.cpp File Reference

`#include "../include/ConceptClass.h"`

Include dependency graph for ConceptClass.cpp:



15.36 ConceptClass.cpp

```
#include "../include/ConceptClass.h"

ConceptClass::ConceptClass(int a, int b) {
    member_a = a;
    member_b = b;
}
```

15.37 learningCpp/OOP/constants.h File Reference

Namespaces

- constants

Variables

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

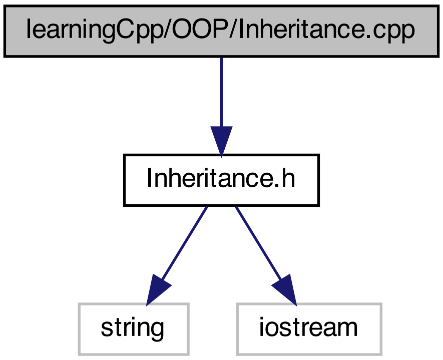
15.38 constants.h

```
//
// Created by Michael Staneker on 01.12.20.
//

#ifndef CPP_TEMPLATE_PROJECT_CONSTANTS_H
#define CPP_TEMPLATE_PROJECT_CONSTANTS_H

namespace constants {

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

    //extern const double pi { 3.141519};
    //extern const double avogadro { 6.0221413e23 };

    // C++17 or newer
    //inline constexpr double pi { 3.14159 }; // inline constexpr is C++17 or newer only
    //inline constexpr double avogadro { 6.0221413e23 };

    //#include "constants.h"
    //
    //double circumfence { 2.0 * radius * constants::pi}
}

#endif //CPP_TEMPLATE_PROJECT_CONSTANTS_H
```

15.39 learningCpp/OOP/Inheritance.cpp File Reference

`#include "Inheritance.h"`

Include dependency graph for Inheritance.cpp:



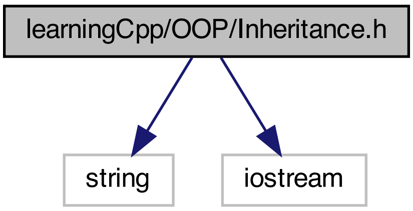
15.40 Inheritance.cpp

```
//
// Created by Michael Staneker on 10.12.20.
//

#include "Inheritance.h"

Base::Base(int id, int var_private, int var_protected, int var_public)
        : m_id{ id }, m_private{ var_private }, m_protected{ var_protected },
          m_public{ var_public }
{
    std::cout « "Base constructor called ..." « std::endl;
}

int Base::getId() const {
    return m_id;
}

int Base::getPrivate() const {
    std::cout « "getPrivate() from Base" « std::endl;
    return m_private;
}

void Base::print() {
    std::cout « "Print from Base class!" « std::endl;
}



Derived::Derived(double cost, int id, int var_private, int var_protected, int var_public)
        : Base{ id, var_private, var_protected, var_public }, // Call Base(int) constructor with value id!
          m_cost{ cost }
{
    std::cout « "Derived constructor called ..." « std::endl;
}

double Derived::getCost() const {
    return m_cost;
}

double Derived::getProtected() const {
    return m_protected;
}

double Derived::getPrivate() const {
    std::cout « "getPrivate() from Derived" « std::endl;
    return Base::getPrivate();
}

void Derived::print() {
    std::cout « "Print from Derived class!" « std::endl;
}
```

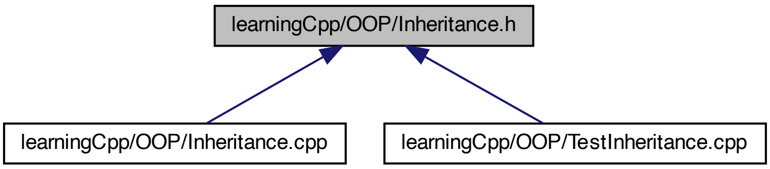
15.41 learningCpp/OOP/Inheritance.h File Reference

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

Include dependency graph for Inheritance.h:



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


learningCpp/OOP/Inheritance.h
learningCpp/OOP/Inheritance.cpp
learningCpp/OOP/TestInheritance.cpp

Classes

- class Base
- class Derived

15.42 Inheritance.h

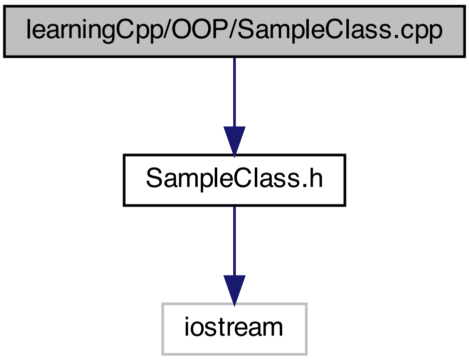
```
//
// Created by Michael Staneker on 10.12.20.
//

#ifndef CPP_TEMPLATE_PROJECT_INHERITANCE_H
```

```
#define CPP_TEMPLATE_PROJECT_INHERITANCE_H

#include <string>
#include <iostream>

class Base
{
private:
    int m_id;
    int m_private;
protected:
    int m_protected;
    int getPrivate() const;
public:
    int m_public;

    Base(int id=0, int var_private=0, int var_protected=0, int var_public=0);

    int getId() const;

    virtual void print();
};

class Derived: public Base
{
private:
    double m_cost;

public:
    Derived(double cost=0.0, int id=0, int var_private=0, int var_protected=0, int var_public=0);
    double getCost() const;
    double getProtected() const;
    double getPrivate() const;

    virtual void print();
};

#endif //CPP_TEMPLATE_PROJECT_INHERITANCE_H
```

15.43 learningCpp/OOP/SampleClass.cpp File Reference

`#include "SampleClass.h"`

Include dependency graph for SampleClass.cpp:



Functions

- void friend_function (SampleClass &sample_class)
- SampleClass operator+ (const SampleClass &s_1, const SampleClass &s_2)
- std::ostream & operator<< (std::ostream &out, const SampleClass &sample_class)

15.43.1 Function Documentation

15.43.1.1 friend_function()

```
void friend_function (
            SampleClass & sample_class )
```

Definition at line 73 of file SampleClass.cpp.

```
00073                                             {
00074     std::cout « "This is a friend function" « std::endl;
00075     std::cout « "Accessing private member member_a: " « sample_class.member_a « std::endl;
00076 }
```

15.43.1.2 operator+()

```
SampleClass operator+ (
            const SampleClass & s_1,
            const SampleClass & s_2 )
```

Definition at line 78 of file SampleClass.cpp.

```
00078                                                                {
00079     std::cout « "overloaded operator+ for SampleClass!" « std::endl;
00080     return SampleClass(s_1.member_a + s_2.member_a, s_1.member_b + s_2.member_b);
00081 }
```

15.43.1.3 operator<<()

```
std::ostream& operator<< (
            std::ostream & out,
            const SampleClass & sample_class )
```

Definition at line 83 of file SampleClass.cpp.

```
00083                                                                    {
00084
00085     out « std::endl
00086         « "member_a = " « sample_class.member_a « std::endl
00087         « "member_b = " « sample_class.member_b « std::endl;
00088
00089     return out;
00090 }
```

15.44 SampleClass.cpp

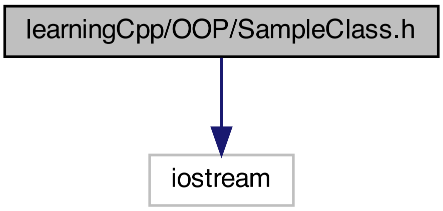
```
//
// Created by Michael Staneker on 09.12.20.
//

#include "SampleClass.h"

SampleClass::SampleClass() {
    std::cout « "Default constructor was called ..." « std::endl;
    member_a = 0;
    member_b = 0;
}

int SampleClass::static_member_variable = 5;

//SampleClass::SampleClass(int a, int b) {
//    std::cout « "Constructor: SampleClass(" « a « ", " « b « ") ..." « std::endl;
//    member_a = a;
//    member_b = b;
//}
// equivalent implementation using an initialization list
SampleClass::SampleClass(int a, int b)
: member_a{ a }, member_b{ b }
{
    std::cout « "Constructor: SampleClass(" « a « ", " « b « ") ..." « std::endl;
}

SampleClass::SampleClass(int a, int b, int c): SampleClass{ a, b } {
    std::cout « "Constructor: SampleClass(" « a « ", " « b « ", " « c « ") ..." « std::endl;
}

```

```
//Destructor
SampleClass::~SampleClass() {
    std::cout « "Destructor was called" « std::endl;
}

SampleClass::SampleClass(const SampleClass &sample_class) :
        member_a(sample_class.member_a), member_b(sample_class.member_b)
{
    std::cout « "Copy constructor called\n"; // just to prove it works
}

int SampleClass::get_member_a() {
    return member_a;
}

int SampleClass::get_member_b() {
    return member_b;
}

void SampleClass::set_member_a(int a) {
    std::cout « "set member_a to: " « a « std::endl;
    member_a = a;
}

void SampleClass::set_member_b(int b) {
    std::cout « "set member_b to: " « b « std::endl;
    member_b = b;
}

void SampleClass::set_members_using_this(int member_a, int member_b) {
    this->member_a = member_a;
    this->member_b = member_b;
}

void SampleClass::const_member_function() const {
    std::cout « "This is a const member function!" « std::endl;
}

void SampleClass::static_member_function() {
    std::cout « "This is a static member function" « std::endl;
}

void friend_function(SampleClass &sample_class) {
    std::cout « "This is a friend function" « std::endl;
    std::cout « "Accessing private member member_a: " « sample_class.member_a « std::endl;
}

SampleClass operator+(const SampleClass &s_1, const SampleClass &s_2) {
    std::cout « "overloaded operator+ for SampleClass!" « std::endl;
    return SampleClass(s_1.member_a + s_2.member_a, s_1.member_b + s_2.member_b);
}

std::ostream& operator« (std::ostream &out, const SampleClass &sample_class) {

    out « std::endl
        « "member_a = " « sample_class.member_a « std::endl
        « "member_b = " « sample_class.member_b « std::endl;

    return out;
}

int SampleClass::operator()(int i) {
    return (member_a += i);
}
```

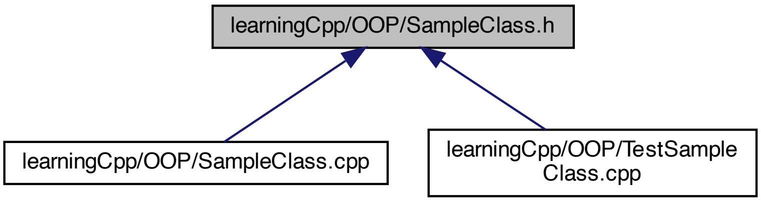
15.45 learningCpp/OOP/SampleClass.h File Reference

`#include <iostream>`

Include dependency graph for SampleClass.h:



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


learningCpp/OOP/SampleClass.h
learningCpp/OOP/SampleClass.cpp
learningCpp/OOP/TestSampleClass.cpp

Classes

- class SampleClass

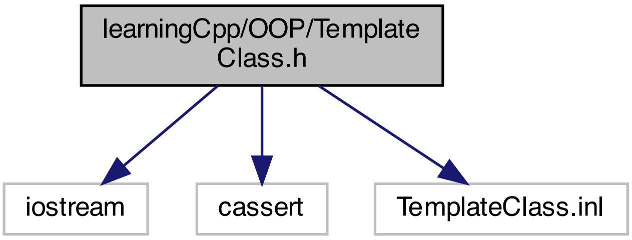
15.46 SampleClass.h

```
//
// Created by Michael Staneker on 09.12.20.
//

#include <iostream>

#ifndef CPP_TEMPLATE_PROJECT_SAMPLECLASS_H
#define CPP_TEMPLATE_PROJECT_SAMPLECLASS_H

class SampleClass {
private:
    int member_a{ 0 };
    int member_b{ 0 };
public:
    //static variables are shared by all objects/instants of the class
    static int static_member_variable;
    // getter
    int get_member_a();
    int get_member_b();
    // setter
    void set_member_a(int a);
    void set_member_b(int b);

    void set_members_using_this(int member_a, int member_b);

    // const member function that cannot change member variables
    // can be overwritten with a non const function
    void const_member_function() const;

    // default constructor
    SampleClass();
    // constructor with arguments
    SampleClass(int a, int b = 0);

    //example for delegating constructors
    SampleClass(int a, int b, int c);

    // Copy constructor
    // prevent copies by making the copy constructor private
    SampleClass(const SampleClass &sample_class);

    ~SampleClass();

    static void static_member_function();

    friend void friend_function(SampleClass &sample_class);

    friend SampleClass operator+(const SampleClass &s_1, const SampleClass &s_2);
    friend std::ostream& operator« (std::ostream &out, const SampleClass &sample_class);

    int operator() (int i);

    // it is possible to have nested types within classes
    enum FruitType {
        APPLE,
        BANANA,
        CHERRY
    };
};



#endif //CPP_TEMPLATE_PROJECT_SAMPLECLASS_H
```

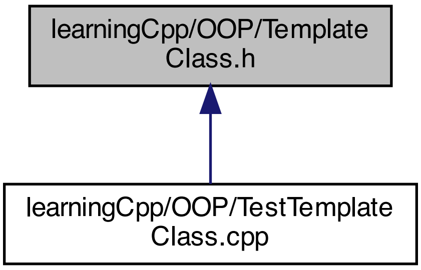
15.47 learningCpp/OOP/TemplateClass.h File Reference

```
#include <iostream>
#include <cassert>
#include "TemplateClass.inl"
```

Include dependency graph for TemplateClass.h:



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



Classes

- class Array< T >
- class StaticArray< T, size >

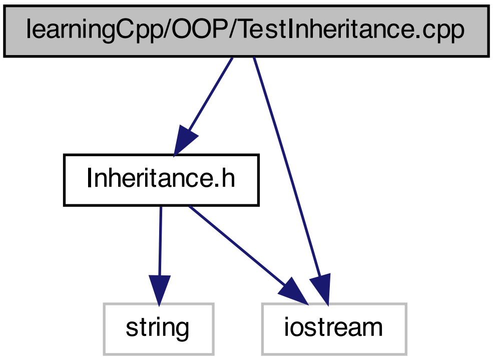
15.48 TemplateClass.h

```
//
// Created by Michael Staneker on 10.12.20.
//

#ifndef CPP_TEMPLATE_PROJECT_TEMPLATECLASS_H
#define CPP_TEMPLATE_PROJECT_TEMPLATECLASS_H

#include <iostream>
#include <cassert>

template <class T>
class Array
{
private:
    int m_length{};
    T *m_data{};
public:

    Array(int length);

    Array(const Array&) = delete;

    Array& operator=(const Array&) = delete;

    ~Array();

    void Erase();

    T& operator[](int index);

    int getLength() const;

    void print();

};


template <class T, int size> // size is the non-type parameter
class StaticArray
{
private:
    // The non-type parameter controls the size of the array
    T m_array[size];

public:
    T* getArray();

    T& operator[](int index);
};

#include "TemplateClass.inl"

#endif //CPP_TEMPLATE_PROJECT_TEMPLATECLASS_H
```

15.49 learningCpp/OOP/TestInheritance.cpp File Reference

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

Include dependency graph for TestInheritance.cpp:



Functions

- int main ()

15.49.1 Function Documentation

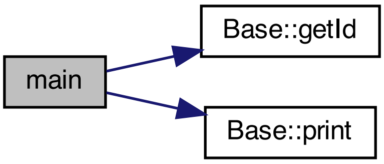
15.49.1.1 main()

```
int main ( )
```

Definition at line 9 of file TestInheritance.cpp.

```
{
    Derived derived{ 1.3, 5, 1, 2, 3};
    std::cout « "Id: " « derived.getId() « '\n';
    std::cout « "Cost: " « derived.getCost() « '\n';
    //std::cout « "private: " « derived.m_private « '\n';
    std::cout « "private: " « derived.getPrivate() « '\n';
    std::cout « "protected: " « derived.getProtected() « '\n';
    std::cout « "public: " « derived.m_public « '\n';

    Base &rBase{ derived };
    derived.print();


    return 0;
}
```

Here is the call graph for this function:



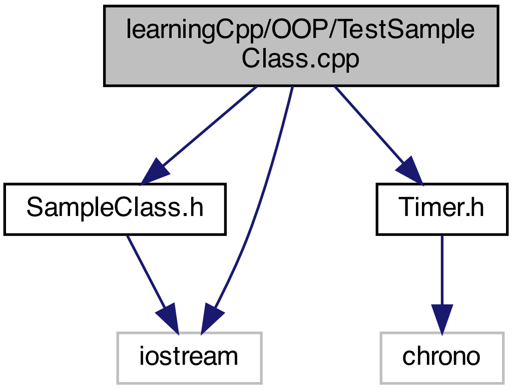
15.50 TestInheritance.cpp

```
//
// Created by Michael Staneker on 10.12.20.
//

#include "Inheritance.h"

#include <iostream>

int main()
{
    Derived derived{ 1.3, 5, 1, 2, 3};
    std::cout « "Id: " « derived.getId() « ’\n’;
    std::cout « "Cost: " « derived.getCost() « ’\n’;
    //std::cout « "private: " « derived.m_private « ’\n’;
    std::cout « "private: " « derived.getPrivate() « ’\n’;
    std::cout « "protected: " « derived.getProtected() « ’\n’;
    std::cout « "public: " « derived.m_public « ’\n’;

    Base &rBase{ derived };
    derived.print();


    return 0;
}
```

15.51 learningCpp/OOP/TestSampleClass.cpp File Reference

```
#include "SampleClass.h"
#include "Timer.h"
#include <iostream>
```

Include dependency graph for TestSampleClass.cpp:



Functions

- int main ()

15.51.1 Function Documentation

15.51.1.1 main()

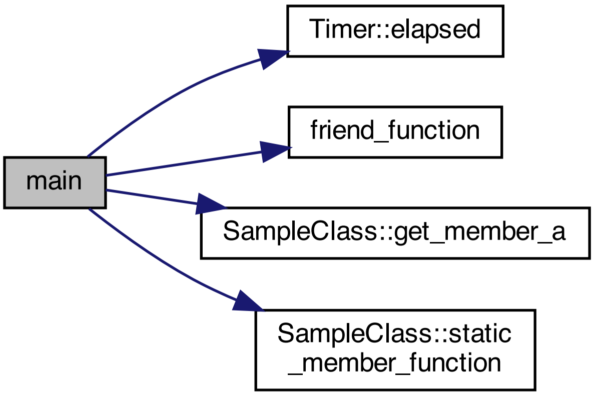
```
int main ( )
```

Definition at line 9 of file TestSampleClass.cpp.

```
00009           {
00010 
00011     Timer t;
00012 
00013     // call default constructor
00014     //SampleClass sample_class;
00015 
00016     // call constructor with arguments
00017     // copy direct initialization
00018     //SampleClass sample_class = SampleClass(5, 10); //equivalent to
00019     // copy list initialization
00020     //SampleClass sample_class = SampleClass{5, 10}; //equivalent to
00021     // direct initialization
00022     //SampleClass sample_class(5, 10); // equivalent to
00023     // list initialization
00024     SampleClass sample_class{5, 10};
00025 
00026     SampleClass copy_sample_class(sample_class);
00027 
00028     // call with default value for member_b
00029     //SampleClass sample_class{5};
00030 
00031     // call with three arguments (ex. for delegating constructors)
00032     //SampleClass sample_class{5, 10, 15};
00033 
00034     std::cout « "sample_class.member_a = " « sample_class.get_member_a() « std::endl;
00035     std::cout « "sample_class.member_b = " « sample_class.get_member_b() « std::endl;
00036 
00037     sample_class.set_member_a(2);
00038     sample_class.set_member_b(4);
00039 
00040     std::cout « "sample_class.member_a = " « sample_class.get_member_a() « std::endl;
00041     std::cout « "sample_class.member_b = " « sample_class.get_member_b() « std::endl;
00042 
00043     // allocate a SampleClass dynamically
00044     SampleClass *dyn_sample_class { new SampleClass{ 1, 6}};
```

```
    std::cout « "dyn_sample_class.member_a = " « dyn_sample_class->get_member_a() « std::endl;
    std::cout « "dyn_sample_class.member_b = " « dyn_sample_class->get_member_b() « std::endl;
    //delete dyn_sample_class;

    // set static counter
    std::cout « "(static) counter = " « SampleClass::static_member_variable « std::endl;
    std::cout « "(static) counter = " « sample_class.static_member_variable « std::endl;

    SampleClass::static_member_function();
    sample_class.static_member_function();

    friend_function(sample_class);

    SampleClass s_1 {2, 2};
    SampleClass s_2 {4, 5};
    SampleClass s_3 = s_1 + s_2;

    std::cout « "s_3: " « s_3 « std::endl;

    std::cout « "s_3(10): " « s_3(10) « std::endl;

    std::cout « "Time taken: " « t.elapsed() « " seconds\n";

    return 0;
}
```

Here is the call graph for this function:



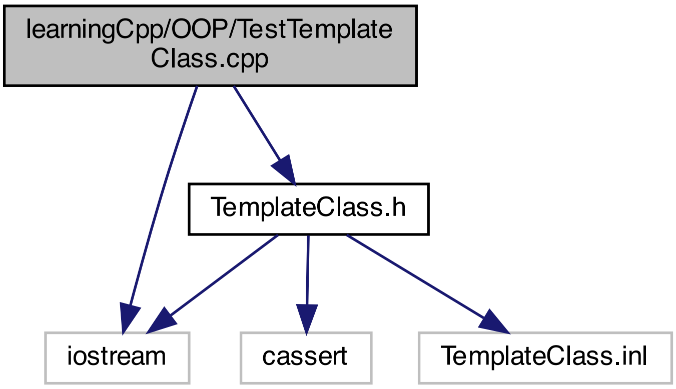
15.52 TestSampleClass.cpp

```
//
// Created by Michael Staneker on 09.12.20.
//

#include "SampleClass.h"
#include "Timer.h"
#include <iostream>

int main() {

    Timer t;

    // call default constructor
    //SampleClass sample_class;

    // call constructor with arguments
    // copy direct initialization
    //SampleClass sample_class = SampleClass(5, 10); //equivalent to
    // copy list initialization
    //SampleClass sample_class = SampleClass{5, 10}; //equivalent to
    // direct initialization
    //SampleClass sample_class(5, 10); // equivalent to
    // list initialization
    SampleClass sample_class{5, 10};

    SampleClass copy_sample_class(sample_class);

    // call with default value for member_b
    //SampleClass sample_class{5};

    // call with three arguments (ex. for delegating constructors)
    //SampleClass sample_class{5, 10, 15};

    std::cout « "sample_class.member_a = " « sample_class.get_member_a() « std::endl;
    std::cout « "sample_class.member_b = " « sample_class.get_member_b() « std::endl;

    sample_class.set_member_a(2);
    sample_class.set_member_b(4);

    std::cout « "sample_class.member_a = " « sample_class.get_member_a() « std::endl;
    std::cout « "sample_class.member_b = " « sample_class.get_member_b() « std::endl;

    // allocate a SampleClass dynamically
    SampleClass *dyn_sample_class { new SampleClass{ 1, 6}};
    std::cout « "dyn_sample_class.member_a = " « dyn_sample_class->get_member_a() « std::endl;
    std::cout « "dyn_sample_class.member_b = " « dyn_sample_class->get_member_b() « std::endl;
    //delete dyn_sample_class;

    // set static counter
    std::cout « "(static) counter = " « SampleClass::static_member_variable « std::endl;
    std::cout « "(static) counter = " « sample_class.static_member_variable « std::endl;

    SampleClass::static_member_function();
    sample_class.static_member_function();

    friend_function(sample_class);

    SampleClass s_1 {2, 2};
    SampleClass s_2 {4, 5};
    SampleClass s_3 = s_1 + s_2;

    std::cout « "s_3: " « s_3 « std::endl;

    std::cout « "s_3(10): " « s_3(10) « std::endl;

    std::cout « "Time taken: " « t.elapsed() « " seconds\n";

    return 0;
}
```

15.53 learningCpp/OOP/TestTemplateClass.cpp File Reference

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

Include dependency graph for TestTemplateClass.cpp:



Functions

- int main ()

15.53.1 Function Documentation

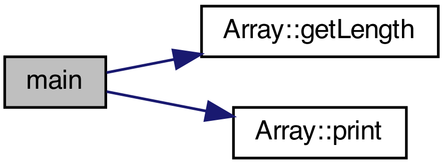
15.53.1.1 main()

```
int main ( )
```

Definition at line 9 of file TestTemplateClass.cpp.

```
{
    Array<int> intArray(12);
    Array<double> doubleArray(12);

    for (int count{ 0 }; count < intArray.getLength(); ++count)
    {
        intArray[count] = count;
        doubleArray[count] = count + 0.5;
    }

    for (int count{ intArray.getLength() - 1 }; count >= 0; --count)
        std::cout « intArray[count] « '\t' « doubleArray[count] « '\n';

    intArray.print();
    doubleArray.print();

    // declare an integer array with room for 12 integers
    StaticArray<int, 12> staticintArray;

    // Fill it up in order, then print it backwards
    for (int count=0; count < 12; ++count)
        staticintArray[count] = count;

    for (int count=11; count >= 0; --count)
        std::cout « staticintArray[count] « " ";
    std::cout « '\n';

    // declare a double buffer with room for 4 doubles
    StaticArray<double, 4> staticdoubleArray;

    for (int count=0; count < 4; ++count)
        staticdoubleArray[count] = 4.4 + 0.1*count;

    for (int count=0; count < 4; ++count)
        std::cout « staticdoubleArray[count] « ' ';

    return 0;
}
```

Here is the call graph for this function:



15.54 TestTemplateClass.cpp

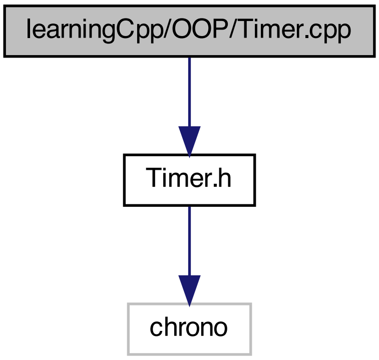
```
//
// Created by Michael Staneker on 10.12.20.
//

#include "TemplateClass.h"
```

```

#include <iostream>

int main()
{
    Array<int> intArray(12);
    Array<double> doubleArray(12);

    for (int count{ 0 }; count < intArray.getLength(); ++count)
    {
        intArray[count] = count;
        doubleArray[count] = count + 0.5;
    }

    for (int count{ intArray.getLength() - 1 }; count >= 0; --count)
        std::cout « intArray[count] « ’\t’ « doubleArray[count] « ’\n’;

    intArray.print();
    doubleArray.print();

    // declare an integer array with room for 12 integers
    StaticArray<int, 12> staticintArray;

    // Fill it up in order, then print it backwards
    for (int count=0; count < 12; ++count)
        staticintArray[count] = count;

    for (int count=11; count >= 0; --count)
        std::cout « staticintArray[count] « " ";
    std::cout « ’\n’;

    // declare a double buffer with room for 4 doubles
    StaticArray<double, 4> staticdoubleArray;

    for (int count=0; count < 4; ++count)
        staticdoubleArray[count] = 4.4 + 0.1*count;

    for (int count=0; count < 4; ++count)
        std::cout « staticdoubleArray[count] « ’ ’;

    return 0;
}

```

15.55 learningCpp/OOP/Timer.cpp File Reference

`#include "Timer.h"`

Include dependency graph for Timer.cpp:



15.56 Timer.cpp

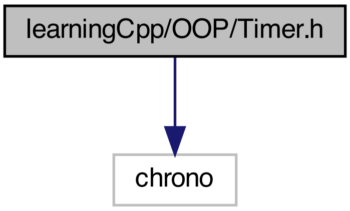
```
//
```

```
// Created by Michael Staneker on 09.12.20.
//

#include "Timer.h"

Timer::Timer() : m_beg(clock_t::now())
{
}

void Timer::reset()
{
    m_beg = clock_t::now();
}

double Timer::elapsed() const
{
    return std::chrono::duration_cast<second_t>(clock_t::now() - m_beg).count();
}
```

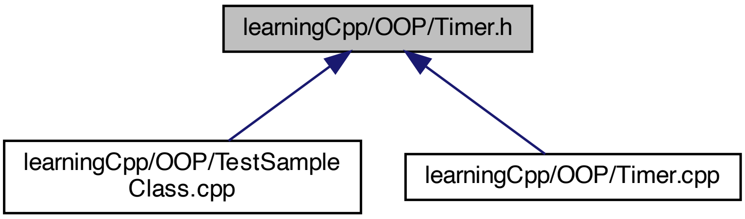
15.57 learningCpp/OOP/Timer.h File Reference

```
#include <chrono>
```

Include dependency graph for Timer.h:



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


learningCpp/OOP/Timer.h
learningCpp/OOP/TestSample Class.cpp
learningCpp/OOP/Timer.cpp

Classes

- class Timer

Macros

- #define CPP_TEMPLATE_PROJECT_TIMER_H

15.57.1 Macro Definition Documentation

15.57.1.1 CPP_TEMPLATE_PROJECT_TIMER_H

```
#define CPP_TEMPLATE_PROJECT_TIMER_H
```

Definition at line 8 of file Timer.h.

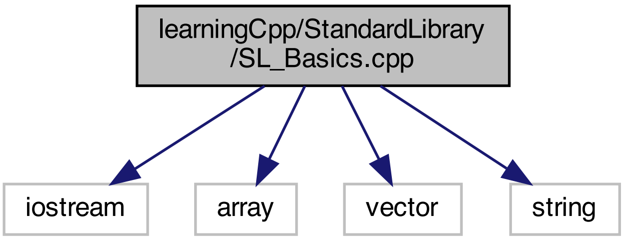
15.58 Timer.h

```
00001 //
00002 // Created by Michael Staneker on 09.12.20.
00003 //
00004
00005 #include <chrono> // for std::chrono functions
00006
00007 #ifndef CPP_TEMPLATE_PROJECT_TIMER_H
00008 #define CPP_TEMPLATE_PROJECT_TIMER_H
00009
00010
00011 class Timer {
00012 private:
00013     // Type aliases to make accessing nested type easier
00014     using clock_t = std::chrono::high_resolution_clock;
00015     using second_t = std::chrono::duration<double, std::ratio<1> >;
00016
00017     std::chrono::time_point<clock_t> m_beg;
00018
00019 public:
00020
00021     Timer();
00022     void reset();
00023     double elapsed() const;
00024
00025 };
00026
00027
00028 #endif //CPP_TEMPLATE_PROJECT_TIMER_H
```

15.59 learningCpp/StandardLibrary/SL_Basics.cpp File Reference

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

Include dependency graph for SL_Basics.cpp:



Functions

- int main ()

15.59.1 Function Documentation

15.59.1.1 main()

```
int main ( )
```

15.59.2 Introduction to the standard library

The Standard library contains a collection of classes that provide templated containers, algorithms, and iterators. If you need a common class or algorithm, odds are the standard library has it. The upside is that you can take advantage of these classes without having to write and debug the classes yourself, and the standard library does a good job providing reasonably efficient versions of these classes. The downside is that the standard library is complex, and can be a little intimidating since everything is templated.

15.59.2.1 STL containers

There are three basic container categories:

- **Sequence containers** maintaining the ordering of elements within the container
  - std::vector: dynamic array capable of growing, fast insertion and removing at the end
  - std::deque: double-ended queue class, implemented as a dynamic array that can grow from both ends
  - std::array
  - std::list
  - std::forward_list
  - std::basic_string
- **Associative containers** automatically sorting the inputs when those inputs are inserted into the container
  - set: storing unique elements
  - mulitset: duplicate elements allowed
  - map (or associative array): each element is a pair, called a key/value pair, key must be unique
  - multimap (or dictionary): map allowing duplicate keys
- **Container adapters**: are special predefined containers that are adapted to specific uses
  - stack: elements operate in a LIFO (Last In, First Out)
  - queue: elements operate in a FIFO (First In, First Out)
  - priority queue: elements are kept sorted (via operator<)

15.59.2.2 STL iterators

- **Operator$*$ $**$ Dereferencing the iterator returns the element that the iterator is currently pointing at**
- $**$**Operator++** Moves the iterator to the next element in the container.
- Most iterators also provide Operator– to move to the previous element.
- Operator== and Operator!= – Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
- Operator= – Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is pointing at, dereference the iterator first, then use the assign operator.

Each container includes four basic member functions for use with **Operator=:**

- *begin()* returns an iterator representing the beginning of the elements in the container. *end()* returns an iterator representing the element just past the end of the elements. *cbegin()* returns a const (read-only) iterator representing the beginning of the elements in the container. *cend()* returns a const (read-only) iterator representing the element just past the end of the elements.

All containers provide (at least) two types of iterators:

- **container::iterator** provides a read/write iterator
- **container::const_iterator** provides a read-only iterator

15.59.2.3 Formatting output

See `ostream`

15.59.2.4 File IO

See `Basic file IO`

15.59.2.5 std::array

15.59.2.6 std::vector

15.59.2.7 std::string

15.59.2.8 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 75 of file SL_Basics.cpp.

```
00075            {
00077     //std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
00078     std::array<int, 5> my_array{9, 7, 5, 3, 1}; // list initialization
00079     my_array[0] = 10; // standard accessing
00080     my_array.at(1) = 8; // other possibility
00081     std::cout « "size of my_array: " « my_array.size() « std::endl;
00085     // dynamic arrays without the need of dynamically allocating memory
00086
00087     //std::vector<int> vec_array;
00088     //std::vector<int> vec_array = { 9, 7, 5, 3, 1 }; // use initializer list to initialize array
       (Before C++11)
00089     std::vector<int> vec_array { 9, 7, 5, 3, 1 }; // use uniform initialization to initialize array
00090     vec_array[0] = 10; // standard accessing
00091     vec_array.at(1) = 8; // other possibility
00092     std::cout « "size of vec_array: " « vec_array.size() « std::endl;
00093     // resize
00094     vec_array.resize(10);
00095     std::cout « "size of vec_array (after resize): " « vec_array.size() « std::endl;
00118     return 0;
00119 }
```

15.60 SL_Basics.cpp

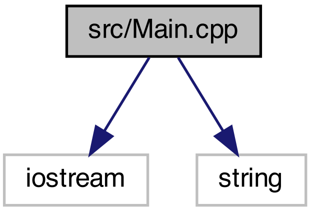
```
00001 //
00002 // Created by Michael Staneker on 03.12.20.
00003 //
00004
00005 #include <iostream>
00006
00007 #include <array> // C++ built in fixed arrays in a safer and more usable form
00008 #include <vector> // makes working with dynamic arrays safer and easier
00009 #include <string> //TODO: section about std::string
00075 int main() {
00077     //std::array<int, 5> myArray = { 9, 7, 5, 3, 1 }; // initializer list
00078     std::array<int, 5> my_array{9, 7, 5, 3, 1}; // list initialization
00079     my_array[0] = 10; // standard accessing
00080     my_array.at(1) = 8; // other possibility
00081     std::cout « "size of my_array: " « my_array.size() « std::endl;
00085     // dynamic arrays without the need of dynamically allocating memory
00086
00087     //std::vector<int> vec_array;
00088     //std::vector<int> vec_array = { 9, 7, 5, 3, 1 }; // use initializer list to initialize array
      (Before C++11)
00089     std::vector<int> vec_array { 9, 7, 5, 3, 1 }; // use uniform initialization to initialize array
00090     vec_array[0] = 10; // standard accessing
00091     vec_array.at(1) = 8; // other possibility
00092     std::cout « "size of vec_array: " « vec_array.size() « std::endl;
00093     // resize
00094     vec_array.resize(10);
00095     std::cout « "size of vec_array (after resize): " « vec_array.size() « std::endl;
00118     return 0;
00119 }
```

15.61 README.md File Reference

15.62 src/Main.cpp File Reference

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

Include dependency graph for Main.cpp:



Functions

- int main ()

15.62.1 Function Documentation

15.62.1.1 main()

```
int main ( )
```

Definition at line 4 of file Main.cpp.

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

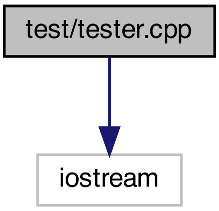
15.63 Main.cpp

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

15.64 test/tester.cpp File Reference

`#include <iostream>`

Include dependency graph for tester.cpp:



Functions

- int main ()

15.64.1 Function Documentation

15.64.1.1 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 }
```

15.65 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
```