OOP for Scientific Computing Notes - SoSe 24

Igor Dimitrov

2024-04-22

Table of contents

| Pr | eface | | | | | | | | | | | | | | | 4 |
|----|------------------------------------|---------------------------------|------|--|--|--|--|--|--|--|---|----|--|---|----|------------------|
| ı | CN | Make Tutorial | | | | | | | | | | | | | | 5 |
| 1 | Step | 0.1 | | | | | | | | | | | | | | 7 |
| | 1.1 | Exercise 1 | | | | | | | | | | | | | | 7 |
| | | 1.1.1 The Three Basic Comm | ands | | | | | | | | | | | | | 7 |
| | | 1.1.2 Getting Started | | | | | | | | | | | | | | 7 |
| | | 1.1.3 Build and Run | | | | | | | | | | | | | | 8 |
| | 1.2 | Exercise 2 | | | | | | | | | | | | | | 9 |
| | | 1.2.1 Getting Started | | | | | | | | | | | | | | 10 |
| | | 1.2.2 Build and Run | | | | | | | | | | | | | | 11 |
| | 1.3 | Exercise 3 | | | | | | | | | | | | | | 12 |
| | | 1.3.1 Getting Started | | | | | | | | | | | | | | 12 |
| | | 1.3.2 Build & Run | | | | | | | | | | | | | | 13 |
| 2 | Step | 2 | | | | | | | | | | | | | 1 | 15 |
| | 2.1 | Exercise 1 - Creating a Library | | | | | | | | | | | | | | 15 |
| | | 2.1.1 Getting Started | | | | | | | | | | | | | | 15 |
| | 2.2 | Exercise 2 - Adding an Option | | | | | | | | | | | | | | 18 |
| | | 2.2.1 Getting Started | | | | | | | | | | | | | | 18 |
| | | 2.2.2 Building & Running | | | | | | | | | | | | | | 21 |
| П | Ba | sic Concepts of C++ | | | | | | | | | | | | | 2 | 23 |
| 3 | 3 Variables, Temporaries, Literals | | | | | | | | | | | | | 2 | 25 | |
| | 3.1 | Variables | | | | | | | | | | | | | | $\frac{1}{25}$ |
| | 3.2 | Temporaries | | | | | | | | | | | | | | 25 |
| | 3.3 | Literals | | | | | | | | | | | | | | 25 |
| 4 | Data Types | | | | | | | | | | 7 | 26 | | | | |
| ٠ | 4.1 | Introducing New Types | | | | | | | | | | | | | | - 0 26 |
| | | 4.1.1 Enum | | | | | | | | | | | | | | $\frac{26}{26}$ |
| | | 4.1.2 Struct | | | | | | | | | | | | | | 26 26 |

| | 4.2 | Const-Correctness | 26 | | | | | | |
|---|--------------------------------------|----------------------------------|----|--|--|--|--|--|--|
| 5 | Indirection | | | | | | | | |
| | 5.1 | Pointers | 28 | | | | | | |
| | 5.2 | References | 29 | | | | | | |
| | 5.3 | Rvalue (double) References | 29 | | | | | | |
| 6 | Control Flow | | | | | | | | |
| | 6.1 | If | 30 | | | | | | |
| | 6.2 | Switch | 30 | | | | | | |
| 7 | Object Orientated Programming in C++ | | | | | | | | |
| | 7.1^{-} | Introduction | 32 | | | | | | |
| | 7.2 | Encapsulation | 33 | | | | | | |
| | 7.3 | | 33 | | | | | | |
| | 7.4 | Constructors & Destructors | 37 | | | | | | |
| | | 7.4.1 Constructors | 37 | | | | | | |
| | | 7.4.2 Destructors | 38 | | | | | | |
| | 7.5 | Pointers / References to Objects | 39 | | | | | | |
| | 7.6 | Composition / Aggregation | 40 | | | | | | |
| | | 7.6.1 Strict Aggregation | 40 | | | | | | |
| | | 7.6.2 Weak Association | 45 | | | | | | |
| | 7.7 | Inheritence | 45 | | | | | | |

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I CMake Tutorial

Notes from the official CMake Tutorial link

1 Step 1

- Introduce CMake basic syntax, commands, and variables.
- Do three exercises and create a simple project.

1.1 Exercise 1

 Most basic CMake project is an executable built from a single file. Only CMakeLists.txt with three components is required. This is our goal with this exercise.

Note

Stylistically lower case commands are preffered in CMake

1.1.1 The Three Basic Commands

- 1. Any project's top most CMakeLists.txt must start by specifying a minimum CMake version using using the cmake_minimum_required() command.
- 2. Afterwards we use the project() command to set the project name.
- 3. Finally we use the add_executable() to make CMake create an executable using the specified source code files

1.1.2 Getting Started

We will build the following c++ file that computes the square root of a number:

• We complete the initial 3 TODOS of the CMakeLists.txt:

Listing 1.1 tutorial.cxx

```
// A simple program that computes the square root of a number
#include <cmath>
#include <cstdlib> // TODO 5: Remove this line
#include <iostream>
#include <string>
// TODO 11: Include TutorialConfig.h
int main(int argc, char* argv[])
 if (argc < 2) {
   // TODO 12: Create a print statement using Tutorial_VERSION_MAJOR
               and Tutorial_VERSION_MINOR
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
   return 1;
 }
 // convert input to double
 // TODO 4: Replace atof(argv[1]) with std::stod(argv[1])
 const double inputValue = atof(argv[1]);
 // calculate square root
 const double outputValue = sqrt(inputValue);
 std::cout << "The square root of " << inputValue << " is " << outputValue
            << std::endl;
 return 0;
```

1.1.3 Build and Run

1. create a build directory:

```
mkdir build
```

2. change into the build directory and build with cmake:

```
cd build
cmake ../
```

3. Actually compile/link the project with

Listing 1.2 CMakelists.txt

```
# TODO 1: Set the minimum required version of CMake to be 3.10
cmake_minimum_required(VERSION 3.10)

# TODO 2: Create a project named Tutorial
project(Tutorial)

# TODO 7: Set the project version number as 1.0 in the above project command

# TODO 6: Set the variable CMAKE_CXX_STANDARD to 11

# and the variable CMAKE_CXX_STANDARD_REQUIRED to True

# TODO 8: Use configure_file to configure and copy TutorialConfig.h.in to

# TODO 3: Add an executable called Tutorial to the project

# Hint: Be sure to specify the source file as tutorial.cxx
add_executable(Tutorial tutorial.cxx)

# TODO 9: Use target_include_directories to include ${PROJECT_BINARY_DIR}
```

```
cmake --build .
```

Now an executable Tutorial has been created and can be run with

./Tutorial 3.0

with the output

The square root of 3 is 1.73205

All good!

1.2 Exercise 2

- CMake has some special variables that have meanig to CMake when set by project
- Many of these variables start with CMAKE_. Two of these special variables:
 - CMAKE_CXX_STANDARD
 - CMAKE_CXX_STANDARD_REQUIRED

- These two together may be used to specify the C++ standard needed to build the project
- Goal: Add a feature that requires C++11 and utilize above two variables. TODO4 TODO6

1.2.1 Getting Started

• TODO 4 & 5 - adding C++11 code to the source tutorial.cxx:

Listing 1.3 tutorial.cxx

```
// A simple program that computes the square root of a number
#include <cmath>
//#include <cstdlib> // TODO 5: Remove this line
#include <iostream>
#include <string>
// TODO 11: Include TutorialConfig.h
int main(int argc, char* argv[])
 if (argc < 2) {
   // TODO 12: Create a print statement using Tutorial_VERSION_MAJOR
               and Tutorial VERSION MINOR
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
   return 1;
 // convert input to double
  // TODO 4: Replace atof(argv[1]) with std::stod(argv[1])
 const double inputValue = std::stod(argv[1]);
 // calculate square root
  const double outputValue = sqrt(inputValue);
 std::cout << "The square root of " << inputValue << " is " << outputValue
            << std::endl;
 return 0;
```

TODO 6 - set the aforementioned variables:

• set(CMAKE_CXX_STANDARD 11)

Listing 1.4 CMakelists.txt

```
# TODO 1: Set the minimum required version of CMake to be 3.10
cmake_minimum_required(VERSION 3.10)

# TODO 2: Create a project named Tutorial
project(Tutorial)

# TODO 7: Set the project version number as 1.0 in the above project command

# TODO 6: Set the variable CMAKE_CXX_STANDARD to 11

# and the variable CMAKE_CXX_STANDARD_REQUIRED to True
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# TODO 8: Use configure_file to configure and copy TutorialConfig.h.in to
# TutorialConfig.h

# TODO 3: Add an executable called Tutorial to the project
# Hint: Be sure to specify the source file as tutorial.cxx
add_executable(Tutorial tutorial.cxx)

# TODO 9: Use target_include_directories to include ${PROJECT_BINARY_DIR}}
```

1.2.2 Build and Run

We already created a build directory adn ran <code>cmake ../</code> in the previous exercise, which created the project configurations. We don't need to redo this steps, instead we simply rebuild the project:

```
cd build
cmake --build .
We run the executable
./Tutorial 10
to obtain:
The square root of 10 is 3.16228
```

1.3 Exercise 3

Sometimes it is useful to have a variable that is defined in CMakelists.txt file also be available in source code. In our case we will define the **version number** in CMakelists.txt and make it available in a header file.

We can accomplished this with a **configured header file**, where there are two variables that can be replaced marked with @VAR@. We use configure_file() command to copy the contents of the configured header file to a standard header file, where the @VAR@ variables are automatically replaced by CMake.

We include this header file generated by CMake in our source code and use the variables defined therein.

We could edit these variables directly in the source code, but using CMake avoids duplication and creates a single source of truth.

Goal: Define and report the project's version number. TODOS: 7 - 12.

1.3.1 Getting Started

First we define the version number with project() command:

```
project(
   Tutorial
   VERSION 1.0
)
```

Now CMake automatically sets in the background two variables:

- Tutorial_VERION_MAJOR as 1
- ullet Tutorial VERION MINOR as 0

since we defined the VERSION as 1.0.

Now we can utilize these variables in a TutorialConfig.h.in file that we will use as an input to CMake to generate a TutorialConfig.h.

We create TutorialConfig.h.in an add following two lines

Note that we access the CMake variables that were previously automatically set by the project() command via the QVARQ syntax.

Next we instruct CMake to generate a TutorialConfig.h from TutorialConfig.h.in with the configure_file() command:

```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

Listing 1.5 TutorialConfig.h.in

```
//File: TutorialConfig.h.in
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
:::
```

The generated header file will be written into the **project binary directory**. In our case it is simply build/ directory.

We must add this directory to the list of paths that CMake searches for include files with the target_include_directories() command:

```
target_include_directories(
  Tutorial
  PUBLIC "${PROJECT_BINARY_DIR}"
)
```

Finally we modify tutorial.cxx to include the generated header file:

```
#include "TutorialConfig.h"
```

and include the print directives that utilize the variables from the header file:

1.3.2 Build & Run

Again we only need to rebuild:

```
cd build cmake --build .
```

If we run **Tutorial** with wrong argument list we get the Version number and the usage message:

./Tutorial

Output:

./Tutorial Version 1.0 Usage: ./Tutorial number

The end!

2 Step 2

- In step 1 we learned how to create a simple project with a single .cxx file and a single executable
- In step 2 we learn:
 - how to create and use a **library**,
 - how to make the use of the library optional

2.1 Exercise 1 - Creating a Library

Goal: Add and use a library

To add a library with CMake, use the add_library() command and specify the source files that make up the library.

Instead of placing all source files in a single directory, we can **organize** our project with one or more subdirectories. Here we create a subdirectory specifically for our library.

To this subdirectory we add another CMakeLists.txt file and source files.

In the top level CMakeLists.txt file, use the add_subdirectory() command to add the subdirectory to the build.

The library is connected to the executable target with

- target include directories()
- target_link_libraries()

2.1.1 Getting Started

We add a library that contains own implementation for computing square root of a number. The executable can then optionally use this library instead of the standard square root function.

The libary is put into a subdirectory MathFunctions. This directory already contains:

• header files:

- mysqrt.h
- MathFunctions.h
- their respective source files:
 - mysqrt.cxx contains custom implementation of square root function
 - MathFunctions.cxx contains a wrapper around sqrt function from msqrt.cxx in order to hide implementation details.
- TODO: 1 6
 - 1. Creating a library target
 - 2. Making use of the new library target
 - 3. Linking the new library target to the executable target
 - 4. Specifying library's header location
 - 5. Using the library
 - 6. Replacing sqrt with the wrapper function mathfunctions::sqrt

In the CMakeLists.txt file in the MathFunctions directory, we craete a library target called MathFunctions with add_library().

TODO 1 - Creating a Library Target

In the CMakeLists.txt in the MathFunctions directory, we create a library target called MathFunctions with add_library():

Listing 2.1 MathFunctions/CMakeLists.txt

TODO 1: Add a library called MathFunctions with sources MathFunctions.cxx add_library(MathFunctions MathFunctions.cxx mysqrt.cxx)

The source files of the library are passed as arguments.

TODO 2 - Making use of the new Library

To make use of the new library we add an add_subdirectory() in the top-level CMakeLists.txt:

Listing 2.2 CMakeLists.txt

add_subdirectory(MathFunctions)

TODO 3 - Linking the new Library Target to the Executable Target

We link the new library target to the executable target with target_link_libraries()

Listing 2.3 CMakeLists.txt

target_link_libraries(Tutorial PUBLIC MathFunctions)

TODO 4 - Specifying Library's Header File Location

Modify the existing target_include_directories() to add the MathFunctions subdirectory as an include directory so that the MathFunctions.h header file can be found:

Listing 2.4 CMakeLists.txt

TODO 5 & 6- Using the Library

We use the library by including MathFunctions.h in tutorial.cxx:

Listing 2.5 tutorial.cxx

```
// TODO 5: Include MathFunctions.h
#include "MathFunctions/MathFunctions.h"
```

Replace sqrt with the wrapper function mathfunctions::sqrt:

Listing 2.6 tutorial.cxx

```
// TODO 6: Replace sqrt with mathfunctions::sqrt

// calculate square root
// const double outputValue = sqrt(inputValue);
const double outputValue = mathfunctions::sqrt(inputValue);
```

2.2 Exercise 2 - Adding an Option

In this exercise we add an option in the MathFunctions library to allow developers to select either the custom or the built-in implementation using the option() command

Goal: Add an option to build without MathFunctions

2.2.1 Getting Started

We will create a variable USE_MYMATH using option() in MathFunctions/CMakeLists.txt There we use that option to pass a compile time definition to the MathFunctions library.

Then, update MathFunctions.cxx to redirect compilation based on USE MYMATH.

Lastly, we prevent mysqrt.cxx from being compiled when USE_MYMATH is on by making it its own library inside of the USE_MYMATH block of MathFunctions/CMakeLists.txt

TODOS: 7 - 14:

- 7. Add an option to MathFunctions/CMakeLists.txt
- 8. Make building and linking our library with mysqrt function conditional using this new option
- 9. Add the corresponding changes to the source code MathFunctions/MathFunctions.cxx
- 10. Including mysqrt.h if the optional varible is defined.
- 11. Including cmath as well
- 12. Ommitting unneccesary usage/build of mysqrt.cxx if the custom option is off.
- 13. Link SqrtLibrary onto MathFunctions when the optional variable is enabled.
- 14. We remove mysqrt.cxx from MathFunctions library source list because it will be pulled when SqrtLibrary is enabled.

TODO 7 - Adding an Option

We add an option to MathFunctions/CMakeLists.txt. This will be displayed in the cmake-gui and ccmake with a default value of ON.

Listing 2.7 MathFunctions/CMakeLists.txt

TODO 7: Create a variable USE_MYMATH using option and set default to ON option(USE_MYMATH "Use custom math implementation" ON)

TODO 8 - Make Building and Linking the Library Conditional

Make building and linking our library with mysqrt function conditional using this new option.

Create an if() statement which checks the value of USE_MATH. Inside the if() put the target_compile_definitions() command with the compile definition USE_MYMATH:

Listing 2.8 MathFunctions/CMakeLists.txt

```
# TODO 8: If USE_MYMATH is ON, use target_compile_definitions to pass
# USE_MYMATH as a precompiled definition to our source files
if(USE_MYMATH)
    target_compile_definitions(MathFunctions PRIVATE "USE_MYMATH")
endif()
```

Now when USE_MYMATH is ON, the compile definition USE_MYMATH will be set. We can then use this compile definitnion to enable or disable sections of our source code.

TODO 9 - Adding the Changes to the Source Code

We add the corresponding changes to the source code. In MathFunctions.cxx we use USE_MYMATH to control which square root function is used:

Listing 2.9 MathFunctions/MathFunctions.cxx

```
// TODO 9: If USE_MYMATH is defined, use detail::mysqrt.
// Otherwise, use std::sqrt.
#ifdef USE_MYMATH
   return detail::mysqrt(x);
#else
   return std::sqrt(x);
#endif
```

TODO 10 - Including mysqrt.h Conditionally

Next, we need to include mysqrt.h if USE_MYMATH is defined.

Listing 2.10 MathFunctions/MathFunctions.cxx

```
// TODO 10: Wrap the mysqrt include in a precompiled ifdef based on USE_MYMATH
#ifdef USE_MYMATH
    #include "mysqrt.h"
#endif
```

TODO 11 - Including cmath

Now since we use std::sqrt() (see TODO 9), we must include cmath:

Listing 2.11 MathFunctions/MathFunctions.cxx

```
// TODO 11: include cmath
#include <cmath>
```

TODO 12 & 13 - Omitting Compilation of mysqrt.cxx if Option is off

At this piont, even if USE_MYMATH is off, mysqrt.cxx would not be used but still compiled because MathFunctions target has mysqrt.cxx listed under sources.

We can fix this in various ways:

- 1. use target_sources() to add mysqrt.cxx rom within the USE_MYMATH block.
- 2. create an additional library within the USE_MYMATH block which is responsible for compiling mysqrt.cxx.

We will go with the second option.

First we create an additional library from within USE_MYMATH called SqrtLibrary that has sources mysqrt.cxx:

Next, we link SqrtLibrary onto MathFunctions when USE MYMATH is enabled:

Listing 2.12 MathFunctions/CMakeLists.txt

Listing 2.13 MathFunctions/CMakeLists.txt

TODO 14 - Removing mysqrt.cxx from Library Source

Finally, we can remove mysqrt.cxx from our MathFunctions library source list because it will be pulled when SqrtLibrary is included.

Listing 2.14 MathFunctions/CMakeLists.txt

```
add_library(MathFunctions MathFunctions.cxx)
```

With these changes, the mysqrt function is now completely optional to whoever is building and using MathFunctions library. Users can toggle USE_MYMATH to this end.

2.2.2 Building & Running

We can manually configure CMake to use the variable providing an option from the command line:

```
cmake ../ -DUSE_MYMATH=OFF #or ON
```

Alternatively we can use cmake-gui or ccmake:

```
ccmake ../
```

and set the automatically detected ${\tt USE_MYMATH}$ variable via the user interface. Afterwards build from within the build directory:

cmake --build .

Part II Basic Concepts of C++

- variables and types
- pointers and references
- control structures
- functions and templates
- classes and inheritance
- $\bullet\,$ name spaces and structure

3 Variables, Temporaries, Literals

- 3.1 Variables
- 3.2 Temporaries
- 3.3 Literals

4 Data Types

4.1 Introducing New Types

4.1.1 Enum

```
enum Color = {RED, BLUE, GREEN}
```

4.1.2 Struct

...

4.2 Const-Correctness

Marks something that can't be modified.

```
include <iostream>
int main(int argc, char const *argv[])
{
   int n = 5;
   const int j = 4;
   const int &k = n; //k can't be modified, equivalently n can't be modified over k
   n++; //but this changes n and indirectly k (because k references n)

const int *p1 = &n; // modifiable pointer to const int
   int const *p2 = &n; // same thing
   int *const p3 = &n; // constant pointer to modifiable int

// p1 = &j -- ok
   // *p1 = 3 -- not ok!
   // p3 = &j -- not ok
   // *p3 = 10 -- ok
```

5 Indirection

5.1 Pointers

```
include <iostream>
int main(int argc, char const *argv[])
    int i = 5;
    int *p1 = &i;
    int *p2 = new int;
    std::cout << "i: " << i << std::endl
              << "*p1: " << *p1 << std::endl
              << "p1: " << p1 << std::endl
              << "&p1: " << &p1 << std::endl
              << "p2: " << p2 << std::endl
              << "*p2: " << *p2 << std::endl;
    delete p2;
    return 0;
output:
i: 5
*p1: 5
p1: 0x7fff8d568184
&p1: 0x7fff8d568188
p2: 0x55c014358eb0
*p2: 0
```

- release memory with delete.
- deleting too early -> bugs, too late -> memory leaks

5.2 References

References are aliases for an existing entity. k

5.3 Rvalue (double) References

Two uses:

- range-based for loops
- move semantics

lvalue references refer to entities, rvalue references refer to literals.

6 Control Flow

6.1 If

```
include <iostream>
int main(int argc, char const *argv[])
{
    int i;
    std::cin >> i;

    if (i % 2 == 0) std::cout << i << " is even" << std::endl;
    else std::cout << i << " is odd" << std::endl;
    return 0;
}</pre>
```

6.2 Switch

```
include <iostream>
enum Color {RED, BLUE, GREEN};
int main(int argc, char const *argv[])
{
   int i;
   Color c = RED;
   std::cin >> i;

   switch(i) {
      case 0:
        c = RED;
      break;
   case 1:
      c = BLUE;
```

```
break;
case 2 :
    c = GREEN;
    break;
default :
    std::cout << "error: invalid color" << std::endl;
}
std::cout << c << std::endl;
return 0;
}</pre>
```

7 Object Orientated Programming in C++

7.1 Introduction

Combine data and functions as a unit. Components of a class are called **members**. In OOP parlance functions members are called **methods**

- methods: behaviour of the object
- data members: state state of the object

Concrete objects created from a class that exist during life-time are called **instances**.

Classes are like new data types, and instance objects are declared just like variables of fundamental types would.

Example

```
class Account
{
    public:
        double get_balance() const;
        double withdraw(double amount);
    private :
        double balance = 100; //initialize with default 100
};

Account::get_balance() {return balance;}
Account::withdraw(doulbe amount) {balance -= amount;}

Account a1;
a1.withdraw(25);
```

i Note

The keyword const in double get_balance() const denotes that get_balance() doesn't modify the state of the object \Rightarrow accessor method.

- accessor methods: methods that do not modify the state of the object like get_balance(). They should be denoted by the const keyword as above.
- mutator methods: methods that modify the state of the object like withdraw().

7.2 Encapsulation

- public members of the class is the interface provided for the user of the class.
- **private** members of the class are used to implement the public interface.

This separation is called **encapulsation** and **information hiding**. This facilitates changing the implementation without changing the interface or affecting other programs that use this class.

Usually an improvement of an existing code comes through changing the underlying data structures. If internal representation if kept hidden from the user of the class, than this change will not effect the user.

Note

It is good practice to list public members before private ones, since users reading the class are primarily interested in the interface as opposed to implementation details.

To adhere to the principle of information hiding **data members** should be always kept **private**. Accessing or mutating the data members should be provided through a public interface of accessor and mutator **member functions**, and never directly.

7.3 Separate Compilation

Encapsulation and information hiding nicely leads to the concept of modularization and separate compilation. When interface definitions and implementations of a class are separated in distinct source files, only the files that are modified can be recompiled, other files need not to be

Consider the situation we want to simulate a cash register machine, with a class that provides the following interface:

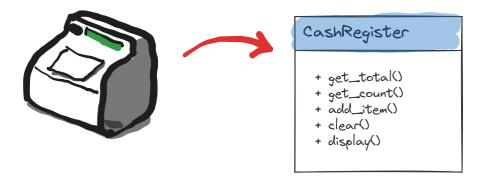


Figure 7.1: cash register machine

We provide the interface for the class 'CashRegister'in a header file:

Listing 7.1 cash_register.hh

```
#ifndef CASHR_H
#define CASHR_H
class CashRegister
{
    public:
        CashRegister();
        double get_total() const;
        int get_count() const;
        void clear();
        void add_item(double amount);
        void display() const;
    private:
        int item_count;
        double price_total;
};
#endif // !CASHR_h
```

Note the

```
#ifdef CASHR_H
#define CASHR_H
//... contents of the header file
```

#endif //CASHR_H

construct. This is called a **header guard**. It is possible that in a project there are many files that use the CashRegister class. When multiple such files are included in another file, the problem will arise that header definitions of CashRegister are included multiple times. As multiple definitions are not legal, this would cause a compiler error. Header guard ensures this, and always should be used.

The implementation of this interface definition is provided separately in a .cpp file:

Listing 7.2 cash_register.cc

```
#include "cash_register.hh"
#include <iostream>
CashRegister::CashRegister()
    item_count = 0;
    price_total = 0;
}
int CashRegister::get_count() const {return item_count;}
double CashRegister::get_total() const {return price_total;}
void CashRegister::display() const {
    std::cout << "count: " << get_count() << std::endl
              << "sum: " << get total() << std::endl;
void CashRegister::clear()
{
    item_count = 0;
    price_total = 0;
}
void CashRegister::add_item(double amount)
{
    item_count++;
    price_total += amount;
}
```

Note that other additional headers needed for the implementation like 'are also included.

Finally we create test program called test_cashregister.cc with a main() function, that will utilize and test the CashRegister class:

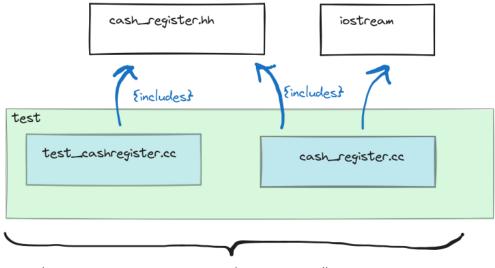
Listing 7.3 test_cashregister.cc

```
#include "cash_register.hh"
#include <iostream>
void display_n(CashRegister cr)
{
    cr.display();
    std::cout << std::endl;</pre>
}
int main(int argc, char const *argv[])
    CashRegister cr1;
    display_n(cr1);
    cr1.add_item(15.4);
    display_n(cr1);
    cr1.clear();
    display_n(cr1);
    return 0;
}
```

Note that test_cashregister.cc only has the interface to CashRegister via the header cash_register.hh but not the actual implementation. This is well intented, since we want to provide the implementation to the compiler as follows:

```
g++ -o test_cashregister test_cashregister.cc cash_register.cc
```

How headers are interrelated and compiled together into an executable can be visualised as follows:



compiled together into an executable program called 'test' via the command: g++ -o test test_register.cc cash_register.cc

Figure 7.2: test CashRegister

The important advantage here is that if the implementation of CashRegister changes, this is reflected solely in cash_register.cc. Thus during recompilation, this file alone needs to be recompiled and linked against test_cashregister.cc. For large software projects and collaborative programming this modularization is essential.

7.4 Constructors & Destructors

7.4.1 Constructors

- Constructor method is called after an object is initialized/created in memory.
- It can defined manually by the programmer, otherwise a default constructor always
 exists. In case of manual definition it can have a list of arguments, just like any other
 method.
- When a constructor is defined manually, the default constructor (one without any arguments must be redefined explicitly by the programmer) ⇒ overloading.
- For class A its constructor is called A(). (Same name as its class)
- Has no return value, but doesn't use keyword void

7.4.2 Destructors

500 400

- The method called before the memory occupied by the object is freed.
- It can be defined by the programmer, otherwise **default destructor** is created.
- Destructor for class A is called ~A()
- Destructors have no arguments, no return value, do not use void.

```
Example:
class Account
    public :
        Account(double amount);
        Account(); //default constructor must be now explicitly defined
        ~Account();
        ... //rest of class
}:
// ... rest of implementations
// initializes account with initial balance of amount
Account::Account(double amount) {balance = amount;}
// overloaded constructors defining default constructor, which
// initializes account to a default value of 100
Account::Account() {balance = 100;}
Then
    Account a1;
    std::cout << a1.get_balance() << std::endl;</pre>
    a1.withdraw(100);
    std::cout << a1.get_balance() << std::endl;</pre>
    Account a2(500);
    std::cout << a2.get_balance() << std::endl;</pre>
    a2.withdraw(100);
    std::cout << a2.get_balance() << std::endl;</pre>
prints out
100
0
```

7.5 Pointers / References to Objects

Pointers or **references** can provide shared access to objects. Assume that a bank account is shared by two people. With pointers:

```
//ap1 points to an account object on heap
Account *ap1 = new Account(300);
//ap2 points to the same object
Account *ap2 = ap1;
```

Visually this looks like:

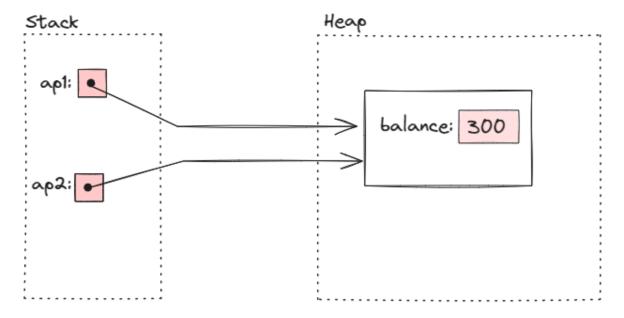


Figure 7.3: object pointers

We can access this objects methods via its pointer:

```
(*ap1).withdraw();
```

Equivalent, and a more common way:

```
ap1->withdraw(10);
```

Which can be understood as: "follow the pointer ap1 to the object it follows and access the method".

The changes will be reflected of course via the pointer ap2:

```
ap1->get_balance()
ap2->get_balance()
//both return the same value of 290

Same can be achieved with references:
Account a(20);
Account &b = a;

//both return 20
a.get_balance();
b.get_balance();

//withdraw 5 from a
a.withdraw(5);

/*changes reflected in both,
both return 15: */
a.get_balance()
b.get_balance()
```

7.6 Composition / Aggregation

In real world objects are usually composed of other objects/components. A car has chasis, tires, engine etc. In oop this is called **composition** or **aggregation**.

The compisition is classified based on the arity of objects that are aggregated and the nature of the association.

The nature of association is divided into two classes:

- strong association/strict aggregation.
- weak association.

First we consider strong association/strict aggregation

7.6.1 Strict Aggregation

In strict aggregation an object is concretely made up of its component objects, and its integrity and lifetime depends on them. A car is strictly composed of four tires and can not exist without the tires.

Strict aggregation itself is categorized depending on the number of objects aggregated:

- Fixed number composition: A car has exactly four tires, or exactly one engine.
- **Arbitrary number composition**: An *order* consists of at least one and at most arbitrary number of *items*.

In UML notation:

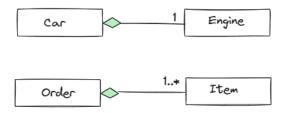


Figure 7.4: aggregation

Fixed Number Aggregation

We first consider the car-engine situation, where a car is composed exactly of one engine. (We ignore all other components that might make up a car for the sake of brevity).

The interface is illustrated in the following UML diagram:

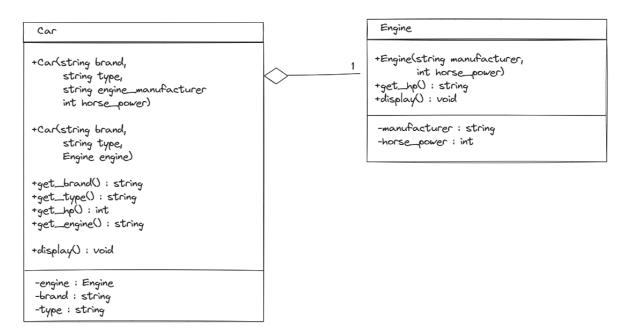


Figure 7.5: car-engine uml

Note that we provide two overloaded constructors: one that receives data members that need to initialize engine component as parameters, and another that receives and engine object as parameter and copies those engines data attributes to the corresponding data attributes in the car object.

c++ implementation with Engine engine data member:

- the interface:
- The implementatin of the classes:
- we test these classes:

output:

brand: Audi
type: sport
engine:

manufacturer: Volkswagen

horse power: 220

brand: Honda
type: family
 engine:

manufacturer: Toyota
horse power: 135

Note

- We used Engine display() function in the implementation of Car's display() function. Thus utilizing 'code-reuse' principle of OOP in the context of aggregation.
- We created two car objects, once with each of the overloaded constructors. Second constructor used an existing engine object to copy the data members of that engine to the corresponding data members in the car object.

Arbitrary Number Aggregation

Now consider the second case where an order **has** at least one item, but can have arbitrary many items.

This situation is demonstrated with the following UML diagram:

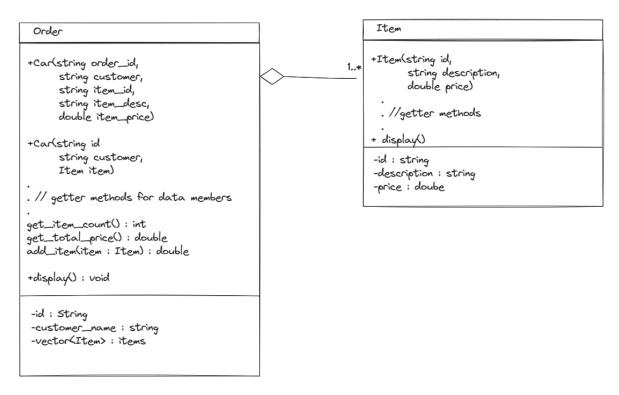


Figure 7.6: order items uml

Note the overloaded constructors for Order.

The c++ implementation with std::vector<Item> items data member:

- the interface:
- the implementation:
- testing:

Finally we compile the test with the command

```
g++ -o test_order test_order.cc order.cc
```

to obtain the output:

order id: 1

customer name: bob

Items:

id: 123

description: water

price: 0.95

item count: 1
total price: 0.95

order id: 1

customer name: bob

Items:

id: 123

description: water

price: 0.95

id: 2

description: bread

price: 1.25

item count: 2
total price: 2.2

order id: 3

customer name: alice

Items:

id: 3

description: cola

price: 2.5

item count: 1
total price: 2.5

id: 2

description: bread

price: 1.25

We created two orders, demonstrating each one of the overloaded constructors, by passing an existing item object to the second order objects constructor as argument.

We demonstrated the display() function both of the Order and of Item objects.

Note that as before, display() function from Order class utilizes code-reuse by using display() from the Item object it aggregates.

The central notion of this chapter is arbitrary number aggregation. An order may have an arbitrary number of items. In c++ it is very common to realize such a relationship with the std::vector<> class. We defined an std::vector<Item> items attribute as a data member of the Order class. Then items simply aggregates Item objects during the lifetime of a given Order object. Since a c++ vector is a dynamic array and therefore can grow arbitrarily, this suitable reflects our sitation.

The constraint that an order must have at least on item is realized by our constructors: there is no defualt constructor, the constructors that we defined initialize an Order object with one Item object. Thus when an ordes is created, it always has one item. Afterwards new items may be added with the add_item() method



🕊 Tip

To implement arbitrary number aggregation use std::vector<> as a data member. E.g. if A aggregates arbitrary number of B s use std::vector collection_of_Bs as a data member of A

7.6.2 Weak Association

7.7 Inheritence

Listing 7.4 car.hh

```
#ifndef CAR_H
#define CAR_H
#include <string>
class Engine
{
    public:
        Engine(std::string manufacturer, int horse_power);
        int get_hp();
        std::string get_brand();
        void display();
    private:
        std::string manufacturer;
        int horse_power;
};
class Car
    public:
        Car(std::string brand,
            std::string type,
            std::string engine_manufacturer,
            int horse_power);
        Car(std::string brand,
            std::string type,
            Engine engine);
        std::string get_brand();
        std::string get_type();
        int get_hp();
        std::string get_engine();
        void display();
    private:
        Engine engine;
        std::string brand;
        std::string type;
};
#endif // !CAR_H
```

Listing 7.5 car.cc

```
#include "car.hh"
#include <iostream>
Engine::Engine(std::string _brand, int _hp) :
    manufacturer(_brand),
    horse_power(_hp) {}
std::string Engine::get_brand() {return manufacturer;}
int Engine::get_hp() {return horse_power;}
void Engine::display()
{
std::cout << "manufacturer: " << get_brand() << std::endl</pre>
          << "horse power: " << get_hp() << std::endl;
}
Car::Car(std::string _brand,
    std::string _type,
    std::string engine_manufacturer,
    int hp) :
        brand(_brand),
        type(_type),
        engine(engine_manufacturer,
                hp) {}
Car::Car(std::string _brand,
         std::string _type,
         Engine _engine) :
            brand(_brand),
            type(_type),
            engine(_engine) {};
std::string Car::get_brand(){return brand;}
std::string Car::get_type() {return type;}
std::string Car::get_engine() {return engine.get_brand();}
int Car::get_hp() {return engine.get_hp();}
void Car::display()
{
std::cout << "brand: " << get_brand() << std::endl
          << "type: " << get_type() << std::endl
          << " engine: " << std::endl;
          engine.display();
}
```

Listing 7.6 test_car.cc

```
#include "car.hh"
#include <iostream>

int main(int argc, char const *argv[])
{
    Car c1("Audi", "sport", "Volkswagen", 220);
    c1.display();
    std::cout << std::endl;

    Engine e("Toyota", 135);
    Car c2("Honda", "family", e);
    c2.display();
    std::cout << std::endl;

    return 0;
}</pre>
```

Listing 7.7 order.hh

```
#ifndef ORDR_H
#define ORDR_H
#include <string>
#include <vector>
class Item
{
    public:
        Item(std::string id,
            std::string description,
            double price);
        std::string get_id();
        std::string get_description();
        double get_price();
        void display();
    private:
        std::string id;
        std::string description;
        double price;
};
class Order
    public:
        Order(std::string order_id,
            std::string customer_name,
            std::string item_id,
            std::string item_desc,
            double item_price);
        Order(std::string id,
            std::string customer_name,
            Item item);
        std::string get_order_id();
        std::string get_customer_name();
        double get_total_price() const;
        int get_item_count() const;
        void add_item(Item item);
                                       49
        void display() const;
    private:
        std::string id;
        std::string customer_name;
        std::vector<Item> items;
};
#endif // !ORDR_H
```

```
#include "order.hh"
#include <iostream>
Item::Item(std::string _id,
    std::string _desc,
    double _price) :
    id(_id),
    description(_desc),
    price(_price) {}
std::string Item::get_id() {return id;}
std::string Item::get_description() {return description;}
double Item::get_price() {return price;}
void Item::display()
    std::cout << "id: " << id << std::endl
              << "description: " << description << std::endl
              << "price: " << price << std::endl;
}
Order::Order(std::string order_id,
        std::string _customer_name,
        std::string item_id,
        std::string item_desc,
        double item_price) :
    id(order_id),
    customer_name(_customer_name)
{
    items.push_back({item_id, item_desc, item_price});
}
Order::Order(std::string _id,
    std::string _customer_name,
    Item item) :
    id(_id),
    customer_name(_customer_name)
{
    items.push_back(item);
}
std::string Order::get_order_id() {return id;}
std::string Order::get_customer_name()<sup>50</sup>(return customer_name;)
double Order::get_total_price() const
    double sum = 0;
    for (auto it : items) {
        sum += it.get_price();
    }
```

Listing 7.9 test_order.cc

```
#include "order.hh"
#include <iostream>
int main(int argc, char const *argv[])
    Order o1(
        "1",
        "bob",
        "123",
        "water",
        0.95
    );
    o1.display();
    Item i1(
        "2",
        "bread",
        1.25
    );
    o1.add_item(i1);
    std::cout << std::endl;</pre>
    o1.display();
    Item i2(
        "3",
        "cola",
        2.5
    );
    Order o2(
        "3",
        "alice",
        i2
    );
    std::cout << std::endl;</pre>
    o2.display();
    std::cout << std::endl;</pre>
    i1.display();
}
```