Lecture 10: Smart pointers and compilation

Jonas Kusch and Martina Prugger

University of Innsbruck

April 28, 2023

Last goals: You are able to

- ☑ define and use abstract classes
- ☑ use templates

Last goals: You are able to

- ☑ define and use abstract classes
- ☑ use templates

Today's learning goals: Today we will learn about

- ☐ smart pointers
- ☐ header files
- ☐ makefiles
- ☐ cmake

Last goals: You are able to

- ☑ define and use abstract classes
- ☑ use templates

Today's learning goals: Today we will learn about

- ☐ smart pointers
- ☐ header files
- ☐ makefiles
- ☐ cmake

Ask questions any time!

Abstract Classes - recap

```
class Cell{
public:
    virtual double Area() = 0;
};

class Triangle : public Cell{
    virtual double Area() { return 1.0; }
};
```

- It does not make sense to create an object `Cell c`, since a cell must always be a triangle, quadrangle, ...
- Every child of cell needs to have an area.
- You can impose this with a pure virtual function `virtual double Area() = 0`.

Templates - recap

There is an easier solution to this task:

```
template <typename T>
T foo( T a, T b ){
    ...
}

double a, b;
T out = foo<double>(a, b)
```

Templates are

- construction plans for the compiler
- can be used to remove code redundancies, avoid repetition, performance

Templates - recap

Combine two typenames:

```
template <typename T, typename S>
S trafo(const T& a){
    S s(a);
    return s;
}

double a = 0.1;
long double c = trafo<double, long double>(a);
```

- typenames are seperated by a comma
- used when typenames are defined at once

Templates - recap

Use for classes

```
template <typename T>
class Container{
    T data;
    ...
}

Container<int> b;
```

- Sometimes you will see the keyword `class` instead of `typename`.
- same meaning, `typename` can be used in all situations (with C++17 or newer)

Smart pointers

```
template<typename T>
class Pointer{
    T* _p;
public:
    Pointer(double* p) : _p(p) {}
    ~Pointer(){delete _p;}
    T& operator*() { return *_p; }
};

Pointer<double> ptr(new double);
*ptr = 0.1;
```

- smart pointers can be used to handle memory deletion for you
- “Garbage Collection of C++”

Smart pointers - unique

```
#include <iostream>
#include <memory>

int main(){
    std::unique_ptr<double> p1(new double);
    *(p1+10) = 0.123;
    std::unique_ptr<double> p2 = std::move(p1);
    std::cout<<*p2<<" "<<*p1<<std::endl;
    return 0;
}
```

- `unique_ptr` stores unique reference
- pointer to reference can only exist once in memory
- p1 no longer stores address after giving it to p2

Smart pointers - shared

```cpp
#include <iostream>
#include <memory>

int main(){
    std::shared_ptr<double> p1(new double);
    std::shared_ptr<double> p2 = p1;

    std::cout<<*p2<<" "<<*p1<<std::endl;
    std::cout << p1.use_count() << std::endl; // #pointers to address
    return 0;
}
```

- shared_ptr stores shared reference
- different pointers can point to same address
- p1 still stores address (assignment operator allowed)

Your turn - What goes wrong? (smart2.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T data) : _data(data){}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::unique_ptr<double> p(new double);
Container<std::unique_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Your turn - What goes wrong? (smart3.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T data){_data = move(data);}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::unique_ptr<double> p(new double);
Container<std::unique_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Your turn - Does this work? (smart4.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T& data){_data = move(data);}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::unique_ptr<double> p(new double);
Container<std::unique_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Your turn - Does this work? (smart.cpp)

```
template <typename T>
class Container{
    T _data;
public:
    Container(T data) : _data(data){}
    T Get()const{return _data;}
    T& Get(){return _data;}
};

std::shared_ptr<double> p(new double);
Container<std::shared_ptr<double>> c(p);
*(c.Get()) = 0.1;
std::cout<<"out = "<<*(c.Get())<<std::endl;
```

Homework

Exercise

Write your own shared smart pointer class which can store dynamical arrays. Use templates to make your code more flexible.

Multiple files (simple)

- for realistic programs putting everything into one file can make the code quite messy.

main.cpp

```cpp
#include <iostream>
#include "functions.cpp"

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

functions.cpp

```cpp
#include <iostream>

void test(){std::cout<<"Hello from functions.cpp"<<std::endl;}
```

Header files (simpleClass)

- split declaration and definition

```cpp
#include <iostream>

class Print{
public:
    void test();
};

void Print::test(){std::cout<<"Hello from print.cpp"<<std::endl;}

int main(){
    Print p; p.test();
    return 0;
}
```

```
g++ main.cpp
```

Header files (simpleClass)

main.cpp

```
#include <iostream>
#include "print.hpp"
...
```

print.hpp (split declaration and definition)

```
class Print{
public:
    void test();
};
```

print.cpp

```
#include <iostream>
#include "print.hpp"
void Print::test(){std::cout<<"Hello from print.cpp"<<std::endl;}
```

```
g++ main.cpp print.cpp
```

Header files (simpleClass)

main.cpp

```
#include <iostream>
#include "print.hpp"
...
```

print.hpp (split declaration and definition)

```
class Print{
public:
    void test();
};
```

print.cpp

```
#include <iostream>
#include "print.hpp"
void Print::test(){std::cout<<"Hello from print.cpp"<<std::endl;}
```

```
g++ main.cpp print.cpp
```

Issues

→ see *classes_demo*

Preprocessor

```
#ifndef CONTAINER
#define CONTAINER

class Container{
public:
    double Value();
};

#endif
```

- preprocessor modifies files before compilation
- prevents double definitions

Exercise

Write classes `Container` which stores a double and `Pair` which stores two containers. Write getter functions for both classes and use seperate files! Try to reproduce a situation where the preprocessor is needed.

The compiler

- We write code that humans understand but computers can't.
  $\Rightarrow$ translate into machine language
- The main tools are the *compiler* and the *linker*.
- *compiler*: C++ code into machine language file (object file, `main.o`, `main.obj`)
- *linker*: combine obj files and libraries (precompiled code)

```
$ g++ -c hello.cpp
$ g++ -o hello.out hello.o
$ ls
hello.cpp  hello.o  hello.out

```

→ https://godbolt.org/

- `g++ -c main.cpp print.cpp container.cpp`
- `g++ main.o print.o container.o -o run`

The compiler

- We write code that humans understand but computers can't.
  $\Rightarrow$ translate into machine language
- The main tools are the *compiler* and the *linker*.
- *compiler*: C++ code into machine language file (object file, `main.o`, `main.obj`)
- *linker*: combine obj files and libraries (precompiled code)

```
$ g++ -c hello.cpp
$ g++ -o hello.out hello.o
$ ls
hello.cpp  hello.o  hello.out

```

$\rightarrow$ https://godbolt.org/

- `g++ -c main.cpp print.cpp container.cpp`
- `g++ main.o print.o container.o -o run`

The compiler

- We write code that humans understand but computers can't.
  ⇒ translate into machine language
- The main tools are the *compiler* and the *linker*.
- *compiler*: C++ code into machine language file (object file, `main.o`, `main.obj`)
- *linker*: combine obj files and libraries (precompiled code)

```
$ g++ -c hello.cpp
$ g++ -o hello.out hello.o
$ ls
hello.cpp  hello.o  hello.out

```

→ `https://godbolt.org/`

- `g++ -c main.cpp print.cpp container.cpp`
- `g++ main.o print.o container.o -o run`

The compiler

- We write code that humans understand but computers can't.
  $\Rightarrow$ translate into machine language
- The main tools are the *compiler* and the *linker*.
- *compiler*: C++ code into machine language file (object file, `main.o`, `main.obj`)
- *linker*: combine obj files and libraries (precompiled code)

```
$ g++ -c hello.cpp
$ g++ -o hello.out hello.o
$ ls
hello.cpp  hello.o  hello.out

```

$\rightarrow$ `https://godbolt.org/`

- `g++ -c main.cpp print.cpp container.cpp`
- `g++ main.o print.o container.o -o run`

Makefiles

Write a file that takes care of compilation for us. Syntax is always

```
target: run when changes
    commands to be run
```

- use tab, not spaces!

```
main.o: main.cpp
        g++ -c main.cpp
```

→ *makefile* demo

Makefiles

```
run: main.o print.o container.o
	g++ main.o print.o container.o -o run

main.o: main.cpp
	g++ -c main.cpp

print.o: print.cpp
	g++ -c print.cpp

container.o: container.cpp
	g++ -c container.cpp

clean:
	rm *.o run
```

- Write a makefile for your Container-Pair example

Makefiles

```
run: main.o print.o container.o
        g++ main.o print.o container.o -o run

main.o: main.cpp
        g++ -c main.cpp

print.o: print.cpp
        g++ -c print.cpp

container.o: container.cpp
        g++ -c container.cpp

clean:
        rm *.o run
```

- Write a makefile for your Container-Pair example

File structure

- All `*.cpp` files go into `src` folder
- All `*.hpp` files go into `include` folder
- Everything generated by make goes into `build` folder

CMake

- There is a program which will take a look at your source code and generate the corresponding make file for you!

CMakeLists.txt

```
cmake_minimum_required( VERSION 3.16 )
set( CMAKE_CXX_STANDARD 17 )
set( CMAKE_CXX_STANDARD_REQUIRED ON )
```

→ sets minimum requirements

```
project( Container VERSION 0.1.0 LANGUAGES CXX )
```

→ generates your project `Container`

```
add_executable( Container src/main.cpp src/print.cpp src/container.cpp )
```

→ adds source files

```
target_include_directories( Container PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include )
```

→ defines include directory

CMake

CMakeLists.txt

```
add_executable( Container src/main.cpp src/print.cpp src/container.cpp )
```

→ can be replaced by

```
file( GLOB_RECURSE SRC_FILES src/*.cpp )
add_executable( Container ${SRC_FILES})
```

- no `../include` needed!
- run cmake with `cmake -S ../ -B .` from build directory

Exercise

Build your previous example with cmake.

CMake - Libraries

- Let's define the container part of our program as a library which we compile independently.
- folder `ext/container` contains `src` and `include` with container files

CMakeLists.txt

```
add_library( Container_lib STATIC ext/container/src/container.cpp)
target_include_directories( Container_lib
        PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/ext/container/include )
```

→ link program and library via

```
target_link_libraries( Container PUBLIC Container_lib )
```

CMake - external libraries

- Some libraries can be linked easily with `find_package`

CMakeLists.txt

```
find_package( LAPACK REQUIRED )
target_include_directories( Container PUBLIC ${LAPACK_INCLUDE_DIR} )
target_link_libraries( Container PUBLIC Container_lib ${LAPACK_LIBRARIES} )
```