OOP for Scientific Computing Notes - SoSe 24

Igor Dimitrov

2024-04-22

Table of contents

Preface

This is a Quarto book.

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

1 Reading List

1. Big C++:
    - 7. Pointers and Structures
    - 8. Streams
    - 9. Classes
    - 10. Inheritence
    - 13. Advanced C++
    - 14. Linked Lists, Stacks and Queues
    - 15. Sets, maps, and Hash Tables
    - 16. Tree Structures
    - 17. Priority Queues and Heaps
2. C++ Primer
    - 2.2 - 2.6
    - 3
    - 4.10,11
    - 5.6
    - 6 - 16
    - 17.1, 17.5
    - 18
    - 19
3. Tour of C++
4. Programming Principles and Practice Using C++, Bjarne Stroustrup
5. C++ Crash Course - Lospinoso
6. Move Semantics C++
    - 1. the power of movec semantics
    - 2. core features of move semantics
    - 3. move semantics in classes
    - 4. how to benefit from move semantics
    - 9. perfecct forwarding
7. C++ Templates - josuttis
    - 1. function templates
    - 2. class templates

- 3. nontype template parameters
- 4. variadic templates
- 5. tricky basics
- 6. move semantics and `enable_if<>`
- 7. By Value or by Reference?
- 8. Compile-Time Programming
- 9. Using Templates in Practice
- 10. Basic Template Terminology
- 11. Generic Libraries
- 12. Fundamnetals in Depth
- 13. Names in Templates
- 14. Instantiation
- 15. Template Argument Deduction
- 16. Specialization and Overloading
- 18. The Polymorphic Power of Templates

8. The C++ STL

   - 1. About this book
   - 2. Intro to C++ and STL
   - 3. New Language Features
   - 4. General Concepts
   - 5. Utilities
   - 6. STL
   - 7. STL Containers
   - 8. STL Container Members in Detail
   - 9. STL Iterators
   - 10. STL function Objects and Using Lambdas
   - 11. STL Algorithms
   - 12. Special Containers
   - 13. Strings
   - 15. Stream Classes
   - 18. Concurerency

9. Discovering Modern C++

   - 1. C++ Basics: 1.5 - 1.8
   - 2. Classes:
   - 3. Generic Programming
   - 4. Libraries
   - 5. Metaprogramming
   - 6. OOP
   - 7. Scientific Projects

10. Data Structures and Algorithms in C++ - Mark Allen Weiss

11. Data Structures and Problem Solving Using C++ - Mark Allen Weisse
12. Objektorientiertes Programmieren in C++
13. Functional Programming in C++ - Cukic
    1. intro to functioal programming
    2. getting started with functional programming
    3. function objects
    4. creating new functions from old ones
    5. purity: avoiding mutable state
    6. lazy evaluation
    7. ranges
    8. functional data structures
    9. algebraic data types and pattern matching
    10. monads
    11. template metaprogramming
    12. functional design for concurrent systesm
    13. Testing and Debugging

Part I

Exam

2 OOSC++ Exam Study Plan SoSe 25

2.1 Revised 14-Day Study Plan (High-Yield, Lecturer-Aligned)

Day 1–2: OOP Essentials

- Classes, access specifiers, encapsulation
- Constructors, destructors, copy/move
- Inheritance, slicing, polymorphism
- Virtual functions, abstract classes, `override`, `final`
- **Practice:** Class hierarchies, virtual dispatch debugging

---

Day 3: RAII + Smart Pointers

- `unique_ptr`, `shared_ptr`, `weak_ptr`, `make_*` functions
- Lifetime, ownership, dangling pointers
- RAII for exception safety and resource cleanup
- **Practice:** Refactor raw pointer code using RAII

Day 4: Lambda Expressions & Closures

- Lambda syntax, capture lists, mutable
- Closures and `std::function`
- Lambdas in algorithms (`sort`, `for_each`, `transform`)
- **Practice:** Lambdas with custom predicates and function composition

---

Day 5: STL Containers + Iterators + Algorithms

- `vector`, `map`, `unordered_map`, `list`, etc.
- Iterator categories and invalidation rules
- Algorithms: `count_if`, `transform`, `copy_if`, etc.
- **Practice:** Design small generic utilities with STL components

---

Day 6: Move Semantics

- Copy vs. move constructors
- `std::move`, lvalues/rvalues
- Rule of 5 / Rule of 0
- **Practice:** Trace object lifetimes in copy/move-heavy code

---

Day 7: Design Patterns (Modern C++)

- Strategy, Visitor, Factory
- Use of polymorphism and smart pointers in patterns
- When to use composition over inheritance
- **Practice:** Recognize patterns in sample code

Day 8–9: Template Programming & Metaprogramming (Core Block)

- Function/class templates, specialization
- Variadic templates, template recursion
- `constexpr` functions and `if constexpr` branching
- Type traits, `enable_if`, SFINAE
- Concepts and constraints (C++20)
- CRTP and static polymorphism
- **Practice:**

  - Write a compile-time factorial
  - Implement type-based dispatch using `constexpr`
  - Build a simple `enable_if` filtering function

Day 10: Modern C++ Features

- Structured bindings, `std::optional`, `std::variant`
- Ranges and views
- `consteval`, `constinit`
- Modules and filesystem (skim unless specifically emphasized)
- **Focus:** What is used in metaprogramming or shown in slides

---

Day 11: Exceptions + Type System

- Exception handling (`throw`, `try`, `catch`, `noexcept`)
- `assert`, contracts
- Const correctness, type deduction (`auto`, `decltype`)
- References vs pointers, `const T*` vs `T const*`
- **Practice:** Spot exception bugs or type deduction issues

Day 12: Deep Dive – Template Metaprogramming Challenge Day

- Redo CRTP and `enable_if` examples
- Try `constexpr` dispatch on types or algorithms
- **Practice:**
  - `static_assert` with trait logic
  - Write a small type trait
  - Recursively define a compile-time structure (e.g., tuple)

Day 13: Mock Exam Simulation

- Time-limited: solve 4–5 realistic tasks
- Balance between code writing, analysis, bug fixing
- **Self-grade** based on expected outputs/behaviors

Day 14: Final Review + Light Touch

- Revisit your 2 weakest areas
- Redo key examples from metaprogramming and OOP
- Skim SOLID principles and major design slides
- Do not cram anything new — stabilize what you know

---

3 Ex 2021

Exercise	1	2	3	4	5	6	$\sum$
Points	15	10	15	15	10	25	90

3.1 Exercise 1

Give a short answer to the following questions, each no longer than three lines. Provide some context, and focus on the central points like, e.g., advantages and disadvantages of certain constructs.

a) What does the keyword `auto` do, and when should it be used?
b) What are lambda expressions? What are they useful for?
c) Give a simple example of a situation where a memory leak will occur, and how it can be resolved.
d) What is a potential use cause for variable templates?
e) What are lvalues and rvalues, and why is there a distinction?
f) When do you need the keyword `explicit`, and why?
g) Explain what inhertience is and why is it useful.
h) Explain the purpose of template specializations through an example.
i) What are virtual methods, and what are they used for?
j) What is an iterator, and why is this concept important for STL algorithms?

3.2 Exercise 2

Let the number sequence $a_n = a(n)$ be given by the following recurrence relation:

$$a(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + a(n - a(a(n-1))) & \text{if } n > 1 \end{cases}$$

The first few values of of this sequence are

$$1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, ...$$

a) Write a template metaprogram that computes n-th number a_n, when given n.
b) Write an equivalent constant expression, as introduced C++11
c) How could you implement the constant expression differently in C++14? (In words only, no code needed)
d) Looking at the numbers of the sequence above, what does a_n specify?

3.3 Exercise 3

The exercise is about inheritance, dynamic polymorphism, and interface classes:

- Assume you have a class `Matrix` and a class `Vector` already given.
- Assume further that a struct `Statistics` is given, derived frmo a struct `StatisticsBase`
- You are tasked with writing a class `Solver` that uses these classes / structs
- Make sure to use appropriate method and attribute qualifiers and encapsulation for this exercise

Parts of the exercise:

a) Write an abstract base class `SolverBase` with the following functionality:

   - A pure virtual function `solve(const Matrix& m, const Vector& b, Vector& x)`
   - A pure virtual function `statistics()` that returns an object of type `StatisticsBase`

b) Write a derived class `Solver` that

   - holds an object of type `Statistics`, into which data of the solution process will be written.
   - provides the methods `solve` and `statistics`, where `statistics` should return an object of type `Statistics`

You may assume that the default constructors / destructor are sufficient. You don't need to implement an algorithm, just provide a dummy function body (like `// [...]`) where the actual algorithm would be written, and make sure that everything of importance is there (i.e., return types and statements, method qualifiers, etc.)

c) The method `solve` of the `Solver` class has a different return type than its abstract base class. What was the name given for this in the lecture? For this to work, the return types must have two properties, name one.

d) Solver classes have to work for various data types, like, e.g., sparse matrices, block matrices or block vectors. How would you need to change your implementation, so that the class `Solver` works for several different matrix and vector classes?

3.4 Exercise 4

This exercise is about using SFINAE to implement a multiplication operator `a * b` that provides products between matrices and vectors.

Assume that type traits `is_matrix<T>` and `is_vector<T>` exist that check if the given type `T` is a matrix or a vector. In particular:

- The trait `is_matrix<T>` checks if `T` fulfills a certain matrix interface:
  - Has a method template `times` for matrix-matrix products, accepting any matrix type.
  - Has a method template `matvec`for matrix-vector products, accepting any vector type.
  - Exports the number of rows and columns as `T::rows` and `T::cols`, respectively.
- The trait `is_vector<T>` checks if `T` fulfills a certain vector interface:
  - Has a method `scalar_prod` for scalar products that expects a vector of the same type.
  - Exports the number of components as `T::comps`.

Use SFINAE with the provided type traits to write the following variants of the operator:

a) Return the scalar product if the first operand `a` is a vector and the second oerand `b` has the same type as `a`.
b) Perform a matrix-vector product if the first operant `a` is a matrix, the second one `b` is a vector, and the number of columns of `a` coincides with the number of components of `b`
c) Calculate a matrix-matrix product if both operands are of matrix type, and the number of columns of `a` is the number of rows of `b`. How can you specify the correct return type easily?
d) How can you achieve the same goal without SFINAE in the current standards, e.g. C++17 or C++20 (In words only, no code needed)

Note: Type combinations other than those mentioned above are allowed to simply cause compilation errors, of course.

3.5 Exercise 5

Assume that a number of classes for k-th derivatives of some function f is provided, with the zeroth one being just a functor for f itself, and each subsequent one being a functor for a derivative of a certain order.

Given such a function f and a development point x_0, the Taolor polynomial of degree n is

$$T_F(x; x_0) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

where $f^{(k)}$ is the k-th derivative of f. Note the following properties of this polynomial:

- Each term except the first k contains the term $\frac{1}{k}$, contributed by the factorial.
- Each subsequent term contains one addditional copy of $(x - x_0)$.

This means we can reuse intermediate values from different terms, and evaluate the polynomial efficiently alternating between multiplication and addition of values.

Provide an implementation of this Taylor polynomial in the form of a variadic template:

- The first template parameter is f itself, the second is its derivative, and so on.
- The degree n is defined through the number of template parameters.
- The points x_0 and x are provided as normal function arguments.
- Use an additional function argument to count the recursion level k, starting at zero.

Note: In practice, this counter is of course hidden away to provide a clean interface, but you can ignore that here.

3.6 Exericse 6

Write a short *essay* (approx, one page) about smart pointers, their origins, their implementation, and their application. In particular, consider the following points, which will be takne into account during grading:

a) What is the *original problem* that smart pointers attempt to solve?
b) What *general technique* are smart pointers and example of, and what is its working mechanism?
c) Why does this solve the aforementioned problem even when *unexpected errors* occur?
d) How do shared pointers *manage their resoruces*, and how is this usually implemented?
e) What are *advantages* and *disadvantages* of using such smart pointers?