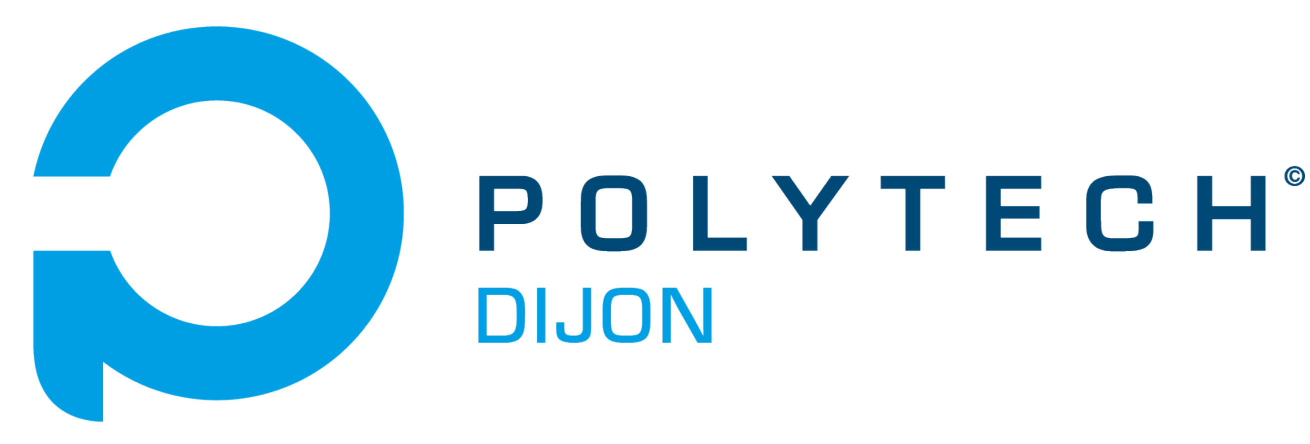


2025/2026

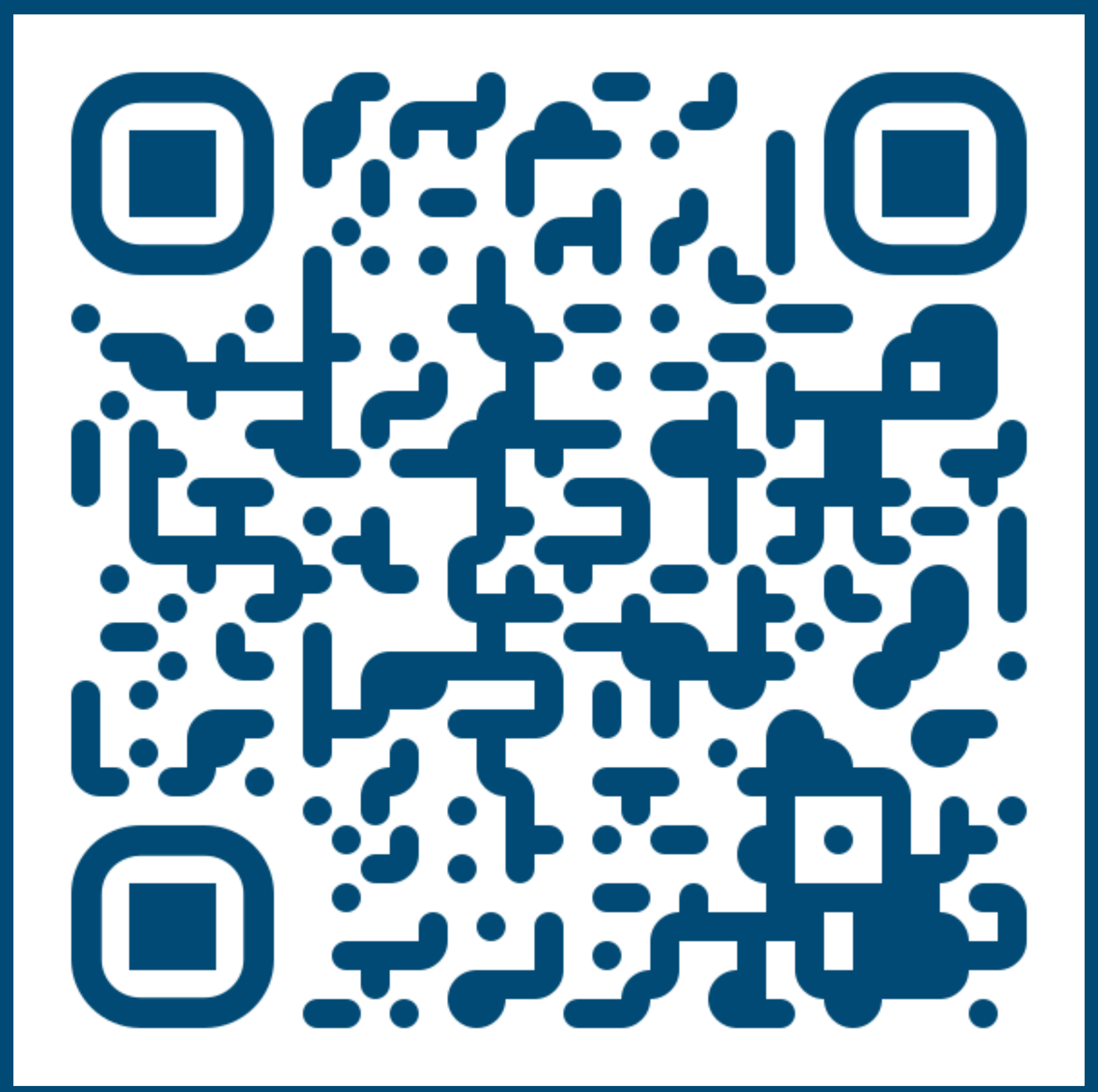
ITC313

Mastering C++:

🎨 The Art of Object-Oriented Programming

Pr. Dominique Ginhac
dginhac@ube.fr

Last updated on Nov, 13, 2025 by dG



🌐 https://ginhac.com/ITC313/02-inheritance.pdf

Lecture #02

https://ginhac.com/ITC313/02-inheritance.pdf

All code samples are available on GitHub in directory "samples/02-inheritance"

Inheritance

Photo by Alex Padurariu on Unsplash

From isolated objects to families of classes

Exploring inheritance — how classes share behavior and structure.

Lecture #00
Course Introduction

Lecture #01
User-defined Data Types
Abstraction/Encapsulation

Today

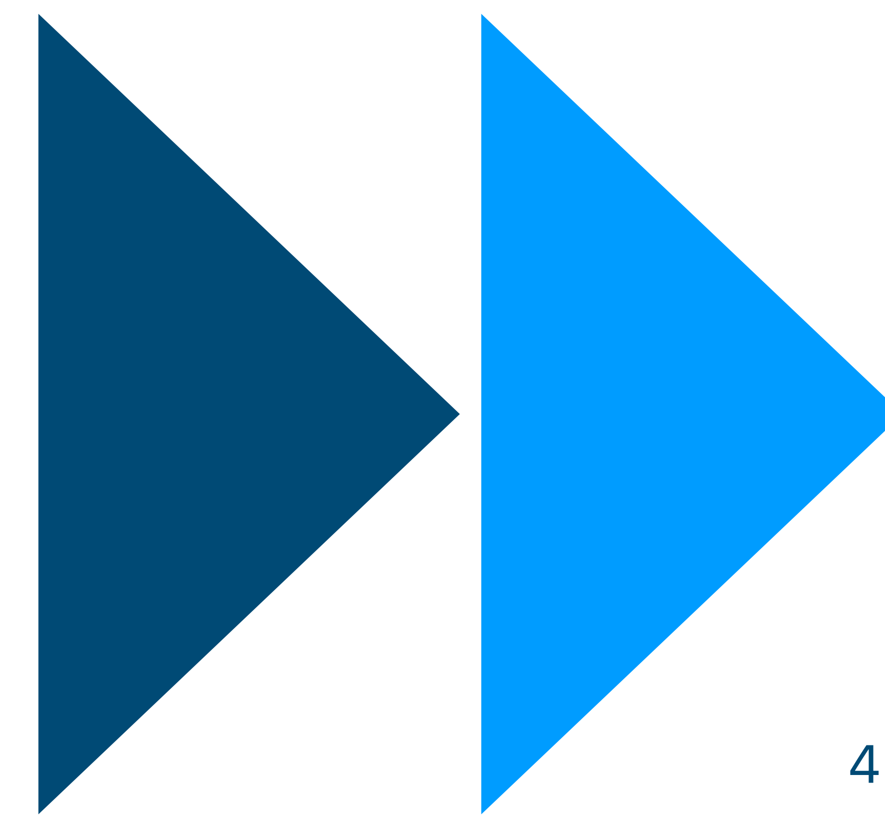
Lecture #02
Inheritance

Lecture #03
Polymorphism

Lecture #04
STL Containers

Lecture #05
Templates

...



AGENDA

02 - Inheritance

1. Understanding inheritance
2. The Todo class

AGENDA

02 - Inheritance

1. Understanding inherit
2. The Todo class

Reminder: the **Four Pillars** of OOP

🏠 **Encapsulation** → Bundle related data and the methods that operate on it into a single, coherent unit — a class.

Ex: a BankAccount has a balance (data) and methods like deposit() or withdraw().

🎭 **Abstraction** → Expose only the essential features, hide the unnecessary details.

Ex: a CoffeeMachine offers brewEspresso() or brewLatte() methods — you don't need to know how water pressure or heating is managed internally.

🧑‍🧑‍🧒 **Inheritance** → Create new classes from existing ones — reuse attributes and behaviors, then extend them.

Ex: an Ebook inherits from a Book, reusing attributes like title and author and adding specific features like url or download().

🔀 **Polymorphism** → One interface (the same function), many forms (depending on the object).

Ex: a Shape offers a draw() method — implemented differently in Circle, Square, and Triangle.

What Inheritance Really Means?

An Ebook inherits from a Book, reusing attributes like `title_` or `author_` and adding specific features like `url_` or `download()`.

🧬 Inheritance creates a **link between two classes** — a parent (Book) and a child (Ebook).

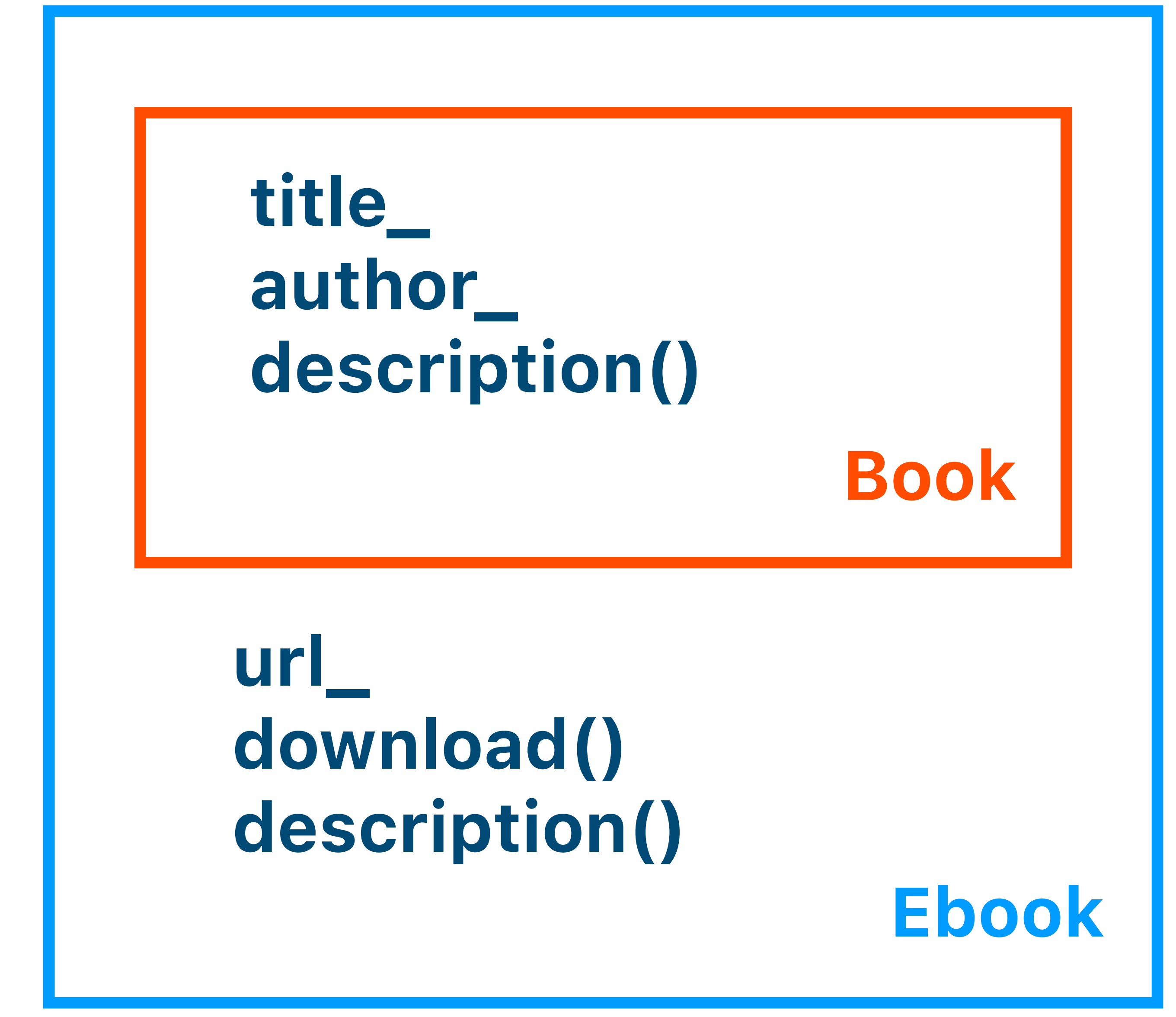
🧠 The child and its parent **share code**.

→ *Avoid **duplication** — Common data like `title_` / `author_` or methods like `description()` are defined once in Book and reused by Ebook.*

⚙️ The child **extends** the parent code

→ *Add new features (like `url_` or `download()`) without the need to rewrite what already works in Book.*

→ *Override parent methods like `description()` to better match the child's behavior.*



Ebook inherits and extends Book.

✨ "Inheritance lets a child class reuse and extend what the parent already defines."

How To Use Inheritance?

🧩 Use the **colon** (:) to declare that one class inherits from another.

→ *class Ebook : public Book { … };*

Choose the **visibility** (public, protected, private) to control which attributes/methods are accessible from the child.

01-Ebook/book.h

```
 1 class Author {
 2 private:
 3     std::string name_;
 4 public:
 5     Author (const std::string& name);
 6     std::string name () const;
 7 };
 8
 9 class Book {
10 private:
11     std::string title_;
12     Author author_;
13 public:
14     Book (const std::string& title, const Author& author);
15     std::string description () const;
16 };
```

01-Ebook/ebook.h

```
 1 #include "book.h"
 2
 3 class Ebook : public Book {
 4 private:
 5     std::string url_;
 6 public:
 7     Ebook (const std::string& title,
              const Author& author, const
              std::string& url);
 8     void download() const;
 9     std::string description () const;
10 };
```

✨ “Inheritance lets a class reuse/update what exists and add what's new — without rewriting.”

Has-A Vs. Is-A Relationship

01-Ebook/book.h

```
class Author {
private:
    std::string name_;
};
class Book {
private:
    std::string title_;
    Author author_;
};
```

01-Ebook/ebook.h

```
#include "book.h"

class Ebook : public Book {
private:
    std::string url_;
public:
    void download() const;
};
```

Composition (Has-a)💬

🧩 A Book **has an** Author → one object owns another.

⚙️ "**Has-a**" describes a part–whole **relationship** (ownership).

Inheritance (Is-a)

🧬 An EBook **is a** Book → one object extends another.

🧭 "**Is-a**" expresses a **type identity** (specialization).

✨ "Both are class relationships, but **composition builds structure** while **inheritance builds identity**."

💬 *"In advanced design, we sometimes distinguish between composition (i.e. strong ownership — the part dies with the whole) and aggregation (i.e. weak ownership — the part can outlive the whole).*

Choosing the Inheritance Type : Public, Protected or Private

🧬 Public inheritance → **“keep it public”** → the child is-a parent.

→ *The public and protected members of the parent keep their visibility.*

→ *The private members are hidden. Use public or protected methods to access them.*

→ *In practice, almost all inheritance is public.*

🛡️ Protected inheritance → **“keep it inside the family”** → inherited members are visible to subclasses.

→ *The child still reuses the parent code, but hides it from the outside.*

🔒 Private inheritance → **“keep it secret”** → all inherited members become private in the child.

→ *The child reuses the implementation, but the parent interface is hidden.*

Member visibility in parent class	Public inheritance	Protected inheritance	Private inheritance
public members	✅ public	🟡 protected	🔒 private
protected members	🟡 protected	🟡 protected	🔒 private
private members	🚫 not inherited	🚫 not inherited	🚫 not inherited

✨ “Inheritance type = **visibility** rule, not behavior.”

Constructors and Inheritance

🧬 When creating a derived object, the base class is built first.

→ *The parent part is constructed before the child part (destruction happens in reverse order — Child first, then parent).*

⚙️ Each constructor only builds its own part.

→ *The base handles its attributes; the derived class handles its own.*

🧩 The derived constructor must provide all arguments

→ *Both for the base class and for the child's own data (you don't build two separate objects — there's only one combined object).*

🧾 Use an initializer list to call the parent's constructor.

→ *DerivedType(args) : BaseType(arg1, arg2, …), … { … }*

01-Ebook/book.cpp

```
1 Author::Author (const std::string& name) : name_(name) {}
2 Book::Book (const std::string& title, const Author& author) :
      title_(title), author_(author) {}
```

01-Ebook/ebook.cpp

```
1 Ebook::Ebook (const std::string& title, const Author&
author, const std::string& url) :
        Book(title, author), url_(url) {}
```

01-Ebook/main.cpp

```
1 int main() {
2     Author author("George Orwell");
3     Book book("1984", author);
4     Ebook ebook("1984", author,
                "http://example.com/1984");
5     return 0;
6 }
```

Methods in Inheritance: Reuse, Redefine, or Extend

🧩 **Reuse** Inherited methods → the child uses parent methods as they are.

→ *Ex: Ebook reuses `Book::author()`.*

⚙️ **Redefine** existing methods → the child overrides parent methods to adapt them to its own behavior.

→ *Ex: `Ebook::description()` adds the URL to the base `Book::description()`.*

✨ **Extend** functionality → the child defines new specific methods, that are not provided by the parent.

→ *Ex: `Ebook::download()`*

01-Ebook/book.cpp

```cpp
Author Book::author() const {
    return author_;
}
std::string Book::description () const {
    return  title_ + " by " + author_.name();
}
```

01-Ebook/ebook.cpp

```cpp
std::string Ebook::description () const {
    return Book::description()
           + " [URL: " + url_ + "]";
}
void Ebook::download() const {
    // Simulate downloading the ebook
    std::cout << "Downloading from "
              << url_ << "..." << std::endl;
}
```

✨ "A derived class can reuse, redefine, or add methods — extending behavior without duplicating code."

It Works, and the Code Is Clean, Readable, and Maintainable

01-Ebook/main.cpp

```cpp
int main() {
    Author author("George Orwell");
    Book book("1984", author);
    Ebook ebook("1984", author, "http://example.com/1984");

    std::cout << "Author: " << ebook.author().name() << std::endl;
    std::cout << "Book: " << book.description() << std::endl;
    std::cout << "Ebook: "<< ebook.description() << std::endl;
    ebook.download();
    return 0;
}
```

```
➜  01-Ebook make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o
➜  01-Ebook ./build/app
Author: George Orwell
Book: 1984 by George Orwell
Ebook: 1984 by George Orwell [URL: http://example.com/1984]
Downloading from http://example.com/1984...
```

👌

Maintainability in Action: Add Features

🧱 Small, local changes only

→ *Updating Author (split `name_` into `firstname_` and `lastname_`) requires no rewrite elsewhere.*

🧩 Encapsulation isolates changes

→ *Only Author's internal structure changes — its public API remains stable for the rest of the code.*

⚙️ Dependencies stay clear and minimal

→ *Book just uses Author's getters in its description().*

→ *Ebook remains fully functional — no modification needed.*

🚀 Design for evolution

→ *Each class focuses on its own responsibility.*

→ *The code adapts easily to new requirements without breaking existing behavior.*

✨ "Well-designed classes make change safe and predictable — modify one, not all."

```cpp
class Author {
private:
    std::string firstname_;
    std::string lastname_;
public:
    Author (const std::string& firstname,
            const std::string& lastname);
    std::string firstname () const;
    std::string lastname () const;
};
```

01b-Ebook/book.cpp

```cpp
std::string Author::firstname () const {
    return firstname_;
}
std::string Author::lastname () const {
    return lastname_;
}
std::string Book::description () const {
    return  title_ + " by " +
            author_.firstname() + " " +
            author_.lastname();
}
```

It Just Works, and the Code Is Still Clean, Readable, and Maintainable

01b-Ebook/main.cpp

```
1  int main() {
2      Author author("George", "Orwell");
3      Book book("1984", author);
4      Ebook ebook("1984", author, "http://example.com/1984");
5
6      std::cout << "Author: " << ebook.author().firstname() << " "
                 << ebook.author().lastname() << std::endl;
7      std::cout << "Book: " << book.description() << std::endl;
8      std::cout << "Ebook: "<< ebook.description() << std::endl;
9      ebook.download();
10     return 0;
```

```
→  01b-Ebook make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o

→  01b-Ebook ./build/app
Author: George Orwell
Book: 1984 by George Orwell
Ebook: 1984 by George Orwell [URL: http://example.com/1984]
Downloading from http://example.com/1984...
```

👌

✨ "When code is well-designed, evolution feels effortless —that's the real proof of maintainability.."

From Inheritance To Design Principles

What we did with Book and Ebook classes was implementing **inheritance** to **reuse** and **specialize** code.

🧩 In software design, this idea becomes a rule: the **OCP** — Open/Closed Principle (#2 of 🏛 SOLID).

OCP: "Classes should be **open for extension** but **closed for modification.**"

✨ Advantages of OCP (in practice):

- 🧱 Ensure **stability** — the base class remains consistent and reliable.
- 👀 Enhance **readability** — extensions are explicit through new subclasses.
- 🧩 Facilitate **evolution** — add new features safely without breaking existing ones.
- 🧠 Reduce **regressions** — prevent accidental changes to proven behavior.
- ⚙️ Improve **maintainability** — no need to modify tested code.

Why OCP Matters

Imagine that you don't create an Ebook class, but instead modify the Book class to handle ebooks.

02-Book/book.h

```cpp
1  class Book {
2  private:
3      std::string title_;
4      Author author_;
5      bool is_ebook_;
6      std::string url_;
7  public:
8      Book (const std::string& title,
             const Author& author,
             const std::string& url = "");
9      std::string description () const;
10     std::string title() const;
11     Author author() const;
12     void download() const;
13 };
```

02-Book/book.cpp

```cpp
1  Book::Book (const std::string& title, const Author& author, const
std::string& url) : title_(title), author_(author), is_ebook_(false),
url_(url) {
2      if (!url.empty()) is_ebook_ = true;
3  }
4  std::string Book::description () const {
5      std::string text = title_ + " by " + author_.name();
6      if (is_ebook_) text += " [URL: " + url_ + "]";
7      return  text;
8  }
9  void Book::download() const {
10     if (!is_ebook_)
11         throw std::logic_error("Not an eBook");
12     std::cout << "Downloading from " << url_ << std::endl;
13 }
```

It Works, but ...

02-Book/main.cpp

```
1  int main() {
2      Author author("George Orwell");
3      Book book("1984", author);
4      Book ebook("1984", author, "http://example.com/1984");
5
6      std::cout << "Author: " << ebook.author().name() << std::endl;
7      std::cout << "Book: " << book.description() << std::endl;
8      std::cout << "Ebook: "<< ebook.description() << std::endl;
9      ebook.download();
10     book.download(); // This will throw an exception
11     return 0;
12 }
```

```
→ 02-Book make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall  -o build/app  build/main.o  build/book.o
→ 02-Book ./build/app
Author: George Orwell
Book: 1984 by George Orwell
Ebook: 1984 by George Orwell [URL: http://example.com/1984]
Downloading from http://example.com/1984
libc++abi: terminating due to uncaught exception of type std::logic_error:
 Not an eBook
[1]    27942 abort      ./build/app
```

...The Code Is Unreadable, Non-Scalable, and Prone to Regressions

🚫 What goes wrong?

🔁 **Trigger endless edits** — adding new types (e.g. Audiobook) means more flags in the existing code.

🧱 **Break stability** — every new change may affect all existing `Book` uses.

📦 **Add irrelevant data** — extra fields (`isEbook`, `url`) are unused for most books.

🧩 **Violate SRP** — `Book` now handles both real books and `download()` logic for ebooks.

⚙️ **Reduce maintainability** — conditional branches (`if (is_ebook)`) spread across code.

✨ "Editing Book to fit ebooks breaks OCP. The base class should stay closed — new features must come from new types."

From Inheritance To Behavioral Integrity

📚 What we did with Book and Ebook was not just inheritance — it was also behavioral integrity, i.e. **preserving consistency** and **respecting rules**.

🧩 In software design, this is a principle called **LSP** — the Liskov Substitution Principle (#3 of 🏛 SO**L**ID).

LSP: "Objects of a subclass should be **substitutable** for objects of the superclass."

Barbara Liskov

✨ LSP in practice:

- 🔒 **Preserve invariants** — keep the parent's internal consistency intact.
- 🧱 **Respect preconditions** — a subclass must not require more to work correctly.
- ✅ **Maintain postconditions** — a subclass must still deliver what the parent guarantees.

B. Liskov et J. Wing, « A behavioral notion of subtyping », ACM Trans. Program. Lang. Syst., vol. 16, no 6, Nov. 1994, p. 1811–1841.
🌐 https://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf

LSP — Preserving the Behavioral Contract

🔒 Preserve **invariants**

→ Invariants are the rules that must always **remain true for an object to stay valid**.

→ A subclass must not break these internal rules of the parent class.

Example: A Book must always have a non-empty title. An EBook must keep that rule true.

⚙️ Respect **preconditions**

→ Preconditions are the conditions that **must be true before a method is called**.

→ A subclass cannot require more than its parent to perform the same action.

Example: If `Book::setTitle()` only forbids empty titles, `Ebook::setTitle()` must not reject short titles — that would strengthen the precondition and break LSP.

✅ Maintain **postconditions**

→ Postconditions are the guarantees that **must be true after a method finishes**.

→ A subclass must still fulfill the promises made by the parent's method.

Example: If `Book::description()` always includes the title and the author, `Ebook::description()` must also include them — it can add information (like the URL) but must not remove what the parent guarantees.

✨ "LSP ensures that subclasses extend behavior without breaking expectations —they must act like their parent, not just look like it.."

Violating LSP #1: Breaking an Invariant

⚙️ A subclass must **not break** valid inputs.

03-Ebook-invariants/book.h

```
1  class Book {
2  protected:
3      std::string title_;
4      Author author_;
5  public:
6      Book (const std::string& title,
             const Author& author);
7      std::string description () const;
8      std::string title() const;
9      Author author() const;
10 };
```

03-Ebook-invariants/book.cpp

```
1  Book::Book (const std::string& title, const
Author& author) : title_(title), author_(author) {
2      if (title.empty()) throw
std::invalid_argument("Title cannot be empty");
3  }
```

03-Ebook-invariants/ebook.cpp

```
1  void Ebook::clearTitle() {
2      title_  = "";
3  };
```

03-Ebook-invariants/main.cpp

```
1  int main() {
2      Author author("George Orwell");
3      Ebook ebook("1984", author,
                   "http://example.com/1984");
4      std::cout << "Ebook: "<< ebook.description()
                 << std::endl;
5      ebook.clearTitle();
6      std::cout << "Ebook: "<< ebook.description()
                 << std::endl;
7      return 0;
8  }
```

```
➜  03-Ebook-invariants make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o

➜  03-Ebook-invariants ./build/app
Ebook: 1984 by George Orwell [URL: http://example.com/1984]
Ebook:  by George Orwell [URL: http://example.com/1984]
```

Encapsulation Protects Invariants

🔒 **Keep parent attributes private — never expose internal state through protected**

→ *Private attributes protect invariants because subclasses can't modify the parent's state directly.*

→ *All changes must go through the parent's API (setters/getters).*

03b-Ebook-invariants/book.h

```
1 class Book {
2 private:
3     std::string title_;
4     Author author_;
5 public:
6     void setTitle(const std::string& title);
7 };
```

03b-Ebook-invariants/ebook.cpp

```
1 void Ebook::clearTitle() {
2     Book::setTitle("");
3 };
```

🛑 Don't add **illegal operations** in a subclass.

If the base class forbids an empty title, a `clearTitle()` method in the child must not exist.

✨ "To preserve invariants → Keep attributes private and control updates through setters."

Violating LSP #2: Strengthening a Precondition

⚙️ A subclass must **not add new restrictions** on valid inputs.

04-Ebook-preconditions/book.cpp

```
1 void Book::setTitle(const std::string& title) {
2     if (title.empty())
          throw std::invalid_argument(
                "Title cannot be empty”);
3     title_ = title;
4 }
```

04-Ebook-preconditions/ebook.cpp

```
1 void Ebook::setTitle(const std::string& title) {
2     if (title.size() < 5)
3         throw std::invalid_argument(
                "Title too short for an eBook");
4     Book::setTitle(title);
5 };
```

04-Ebook-preconditions/main.cpp

```
 1 int main() {
 2     Author author("George Orwell");
 3     Book book("Nineteen eighty four", author);
 4     Ebook ebook("Nineteen Eighty-Four)", author,
                   "http://example.com/1984");
 5     book.setTitle("1984");
 6     std::cout << book.description()
                 << std::endl;
 7     ebook.setTitle("1984");
 8     std::cout << ebook.description()
                 << std::endl;
 9     return 0;
10 }
```

```
→  04-Ebook-preconditions make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o
→  04-Ebook-preconditions ./build/app
1984 by George Orwell
libc++abi: terminating due to uncaught exception of type std::invalid_argument:
 Title too short for an eBook
[1]    85531 abort      ./build/app
```

Preventing Precondition Violations

🚫 **Don't strengthen the parent's rules.**

→ *A subclass must not add new restrictions on valid inputs.*

→ *Making the method "stricter" breaks substitutability.*

Keep validation logic in the parent.

→ *The parent owns input validation and defines what is valid.*

→ *The child reuses it or relaxes it if needed, but never tightens it.*

04b-Ebook-preconditions/book.cpp

```
1 void Ebook::setTitle(const std::string& title) {
2     const std::string cleaned = trim(title);
3     Book::setTitle(cleaned);
4 };
```

🧩 *The `trim()` function cleans the input by removing spaces at the beginning and end of the title — its code is available on the course repository.*

04b-Ebook-preconditions/main.cpp

```
1  int main() {
2      Author author("George Orwell");
3      Book book("Nineteen eighty four", author);
4      Ebook ebook("Nineteen Eighty-Four)", author,
                   "http://example.com/1984");
5      book.setTitle("      1984   ");
6      std::cout << book.description() << std::endl;
7      ebook.setTitle("      1984   ");
8      std::cout << ebook.description()<< std::endl;
9      return 0;
10 }
```

```
→  04b-Ebook-preconditions make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o

→  04b-Ebook-preconditions ./build/app
      1984    by George Orwell
1984 by George Orwell [URL: http://example.com/1984]
```

✨ "To preserve preconditions → Keep validation in the parent and don't restrict inputs in subclasses."

Violating LSP #3: Weakening a Postcondition

⚙️ A subclass must deliver **at least what the parent guarantees** — never less.

05-Ebook-postconditions/book.cpp

```cpp
std::string Book::description () const {
    return  title_ + " by " + author_.name();
}
```

05-Ebook-postconditions/ebook.cpp

```cpp
std::string Ebook::description () const {
    return "[URL: " + url_ + "]";
}
```

⚠️ Ebook::description() breaks the parent's contract — it no longer returns the title and author as Book::description() guarantees.

05-Ebook-postconditions/main.cpp

```cpp
int main() {
    Author author("George Orwell");
    Book book("Nineteen eighty four", author);
    Ebook ebook("Nineteen Eighty-Four", author,
                "http://example.com/1984");
    std::cout << book.description()
              << std::endl;
    std::cout << ebook.description()
              << std::endl;
    return 0;
}
```

```
→  05-Ebook-postconditions make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o
→  05-Ebook-postconditions ./build/app
Nineteen eighty four by George Orwell
[URL: http://example.com/1984]
```

Preventing Postcondition Violations

🚫 **Don't reduce the parent's guarantees.**

→ *A subclass must still deliver what the parent promises.*

→ *It may add new information, but never remove what was expected.*

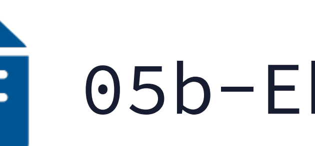
Extend, don't replace, the parent's results.

→ *When redefining a behavior, keep the parent's logic as the foundation.*

→ *The child may enrich it (e.g., add an URL), but not change its meaning.*

05b-Ebook-postconditions/ebook.cpp

```cpp
1 std::string Ebook::description () const {
2     return Book::description()
              + " [URL: " + url_ + "]";
3 }
```

 05b-Ebook-postconditions/main.cpp

```cpp
1 int main() {
2     Author author("George Orwell");
3     Book book("Nineteen eighty four", author);
4     Ebook ebook("Nineteen Eighty-Four", author,
                  "http://example.com/1984");
5     std::cout << book.description() << std::endl;
6     std::cout << ebook.description()
                << std::endl;
7     return 0;
8 }
```

```
→ 05b-Ebook-postconditions make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c book.cpp -o build/book.o
clang++ -Wall -MMD -c ebook.cpp -o build/ebook.o
clang++ -Wall  -o build/app  build/main.o  build/book.o  build/ebook.o
→ 05b-Ebook-postconditions ./build/app
Nineteen eighty four by George Orwell
Nineteen Eighty-Four by George Orwell [URL: http://example.com/1984]
```

✨ "To preserve postconditions → Keep the parent's guarantees and extend its results safely."

LSP — Preserving Behavioral Integrity

1️⃣ LSP holds when you preserve the parent's behavioral contract

→ *Keep **invariants**, **preconditions**, and **postconditions** consistent with the parent class.*

→ *A child must behave like its parent — not just compile like it.*

2️⃣ LSP ensures reliable and predictable inheritance

→ *By keeping the parent's behavioral rules, subclasses can be **used safely** in place of their parent.*

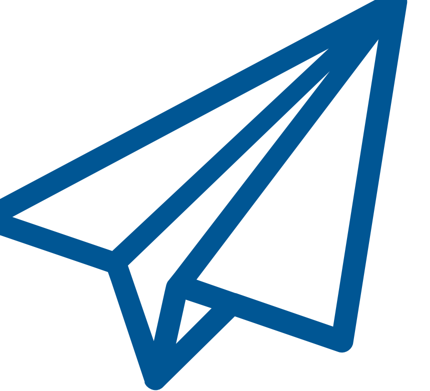
→ *This prevents **unexpected side effects** and makes code **easier to maintain**.*

3️⃣ We'll revisit LSP later

→ When studying **polymorphism** and **abstract classes**, we'll see how LSP ensures that substituting a derived object for its base works correctly at runtime.

✨ "LSP ensures that inheritance is not only structural, but also behavioral. Respect the parent's rules, and substitution will always be safe."

A FIFTH TAKE HOME MESSAGE



Inheritance — what really matters

🧩 **Reuse and specialize** — share common attributes and behaviors, then extend them in child classes (SRP).

🚀 **Extend, don't edit** — add new subclasses to grow functionality without breaking existing code (OCP).

🔒 **Respect encapsulation** — children inherit structure, not private data; use public API access when needed.

🛡️ **Preserve parent** — subclasses must not alter the parent's (LSP).

Questions

AGENDA

02 - Inheritance

1. Understanding inheritance
2. The Todo class

AGENDA

02 - Inheritance

1. Understanding inheritance
2. **The Todo class**

Let's Take a Realistic Use Case — a Todo List App

💡 Imagine you need to code a simple **Todo list manager**.

→ *What are the essential pieces of data for each task?*

🧱 Core attributes of a generic Todo

- 🏷️ ***Title*** → *task name or short description (`std::string`)*
- 📅 ***Due date*** → *when it should be done (`Date`)*
- ✅ ***Status*** → *done or not (`bool`)*
- 🗂️ ***Category*** → *type of task (`enum class Category`)*
- ⚙️ ***Priority*** → *how urgent it is (`int` or `enum class Priority`)*

✨ "Before coding behavior, always define the core data — what every Todo must contain."

The Todo Class

06-Todo/todo.h

```
1  class Todo {
2  public:
3      Todo(const std::string& title, const Date& scheduledDate, Category category, Priority priority);
4      const std::string& title() const;
5      const Date& scheduledDate() const;
6      Category category() const;
7      Priority priority() const;
8      bool isDone() const;
9      void updateTitle(const std::string& title);
10     void updateScheduledDate(const Date& date);
11     void updateCategory(Category category);
12     void updatePriority(Priority priority);
13     void markAsDone();
14 private:
15     std::string title_;
16     Date scheduled_date_;
17     Category category_;
18     Priority priority_;
19     bool done_;
20 };
21 // Utility function (for display)
22 std::string description(const Todo& todo);
```

```
enum class Category {
    Research, Teaching, Personal
};

enum class Priority {
    Low, Medium, High
};
```

🧠 **Todo class models what a task is.**

Todo focuses on data and rules, with methods names that express intent, not implementation.

Using the Todo Class

06-Todo/main.cpp

```cpp
#include "todo.h"

int main() {
    std::string title = "Finish slides on C++";
    Date due_date(12,1);
    Todo todo1(title, due_date, Category::Teaching, Priority::Medium);
    std::cout << description(todo1)
              << std::endl;
    // Update due date and priority
    todo1.updateScheduledDate(Date(12,15));
    todo1.updatePriority(Priority::High);
    std::cout << description(todo1)
              << std::endl;
    // All slides are ready,
    mark as done
    todo1.markAsDone();
    std::cout << description(todo1)
              << std::endl;
    return 0;
}
```

```
→  06-Todo make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c todo.cpp -o build/todo.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/todo.o  build/date.o

→  06-Todo ./build/app
Finish slides on C++ [Pending] – Teaching – Priority Medium – Due: 12/1
Finish slides on C++ [Pending] – Teaching – Priority High – Due: 12/15
Finish slides on C++ [Done] – Teaching – Priority High – Due: 12/15
```

💡 Full implementation of the Todo class is available on the course repository.

From Todo to RecurringTodo — When Inheritance Makes Sense

💡 A recurring Todo = a task that happens on a regular basis

→ *When you mark it complete, a new Todo is automatically generated.*

→ *It repeats based on a defined pattern (daily, weekly, monthly, ...).*

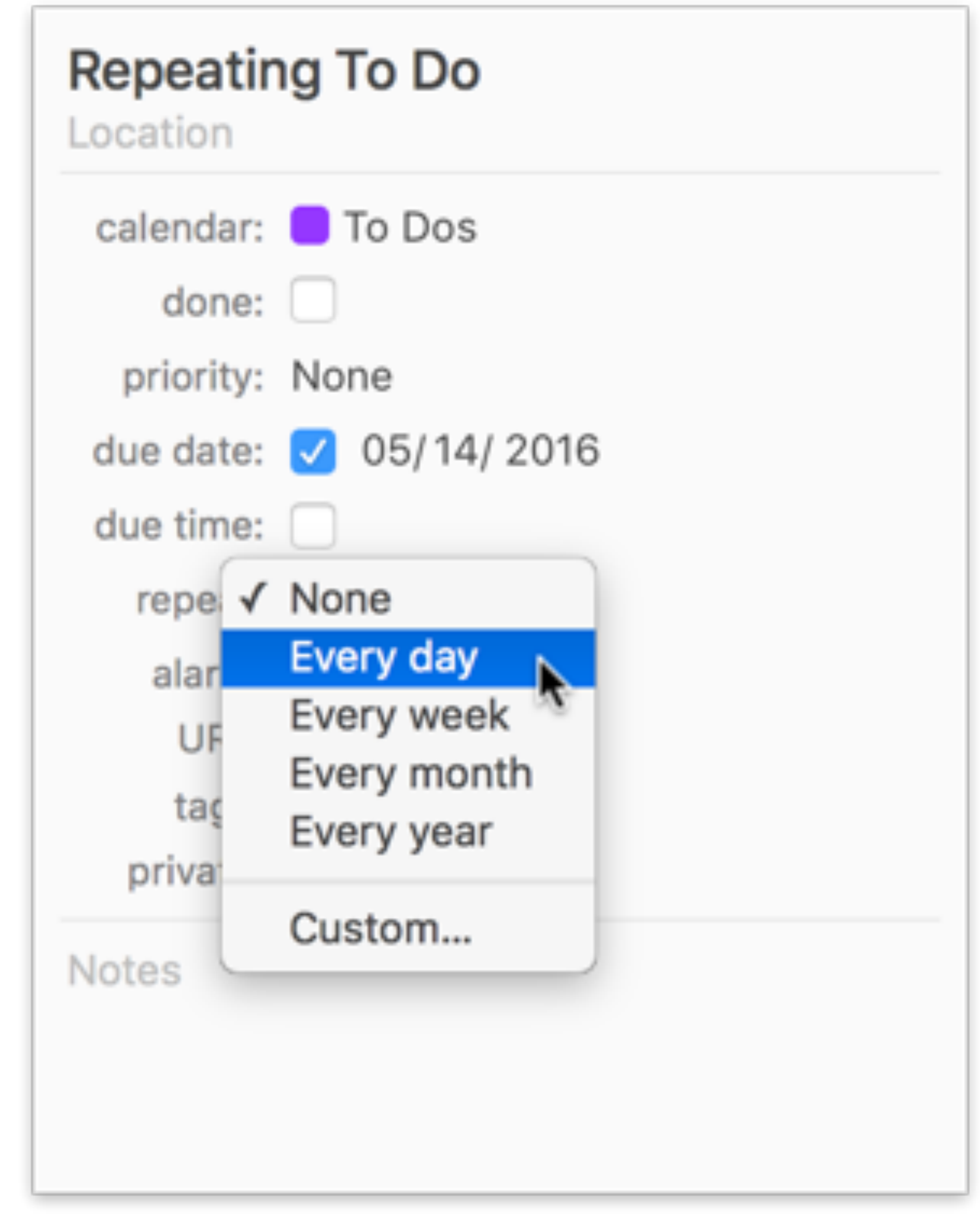
🧱 Similar to a standard Todo

🏷️ *Title,* 📅 *Due date,* ✅ *Status,* 🗂️ *Category and* ⚙️ *Priority.*

✨ But adds new properties

🕓 *End date — when the repetition stops (`Date`)*

🔁 *Period — time interval between occurrences (`int`)*



✨ “The RecurringTodo class keeps the same identity as Todo, but adds new behavior — that’s when **inheritance fits**.”

How RecurringTodo Design Respects SOLID Principles

✅ SRP: Single Responsibility Principle → 🎯 clear, single responsibility per class.

Todo: manage a single task.

RecurringTodo: manage recurrence logic.

✅ OCP: Open/Closed Principle → 🧩 extend behavior without touching base code.

🔓 *Open for extension → new features (recurrence) via inheritance.*

🔒 *Closed for modification → Todo class remains unchanged.*

✅ LSP: Liskov Substitution Principle → 🛡️ reuse safely through strict encapsulation and consistent contracts.

🧩 *RecurringTodo accesses Todo attributes only through the parent's public API (getters / setters).*

⚙️ *All invariants and pre/postconditions from Todo are respected.*

✨ "RecurringTodo enriches Todo (SRP + OCP) while staying fully substitutable (LSP)."

SRP in Action — the RecurringTodo Class

07-RecurringTodo/recurring-todo.h

```cpp
class RecurringTodo : public Todo {
public:
    RecurringTodo(const std::string& title,
                  const Date& scheduled_date,
                  Category category,
                  Priority priority,
                  const Date& end_date,
                  int period_days);
    // Accessors
    const Date& endDate() const;
    int periodDays() const;
    // Mutators
    void updateEndDate(const Date& end_date);
    void updatePeriodDays(int period_days);
    // Behavior specific to recurring todos
    void completeAndScheduleNext();
private:
    Date end_date_;
    int period_days_;
};
```

🧱 Declaration

🧩 *RecurringTodo is defined as a public derived class of Todo.*

🔁 *It inherits all attributes and methods from Todo.*

🧠 New member variables

📅 *end_date_ → when the repetition stops (`Date`).*

⏳ *period_days_ → number of days between two occurrences (`int`).*

→ *All other variables are shared with the base class (`Todo`) and available through the Todo public API (Getters/Setters).*

⚙️ New methods

🪜 Getters / Setters for `end_date_` and `period_days_`.

✅ `completeAndScheduleNext()` → schedules the next occurrence and marks the task as done when `end_date_` is over.

✨ "RecurringTodo extends Todo with two new attributes and one clear behavior — recurrence management — while reusing all the rest."

Constructing a RecurringTodo Object — Do It in the Initializer List

07-RecurringTodo/recurring-todo.cpp

```
 1 RecurringTodo::RecurringTodo(
 2             const std::string& title,
 3             const Date& scheduled_date,
 4             Category category,
 5             Priority priority,
 6             const Date& end_date,
 7             int period_days)
 8     : Todo(title, scheduled_date, category, priority),
 9       end_date_(end_date),
10       period_days_(period_days) {
11     if (period_days_ <= 0) {
12         throw std::invalid_argument(
               "period_days must be positive");
13     }
14     // For a recurring task, the end date
       // should be >= initial scheduled date
15     if (end_date_ < scheduledDate()) {
16         throw std::invalid_argument(
                "end_date must be on or after the
                 scheduled date");
17     }
18 }
```

🧩 Pass all required parameters to RecurringTodo

→ *title, scheduled_date, category, priority, end_date, period_days.*

🚚 Call the base constructor in the initializer list

→ *RecurringTodo(...) : Todo(title, scheduled_date, category, priority), ...*

→ *This is the only correct way to build the base part.*

🧱 Initialize all members in the initializer list

→ *Avoid "default then assign".*

→ *Remember: initialization order follows the declaration order in the class.*

🛡️ Validate recurrence invariants early

→ *period_days > 0 and end_date >= scheduled_date (how to compare Date will be discussed later).*

🎯 Keep the constructor focused

→ *No heavy work; just initialize and validate.*

OCP in Action — the One Method That Defines Recurrence Behavior

07-RecurringTodo/recurring-todo.cpp

```cpp
void RecurringTodo::completeAndScheduleNext() {
    const Date next = scheduledDate() + period_days_;
    if (next <= end_date_) {
        // Series continue : next occurrence scheduled
        updateScheduledDate(next); // reuse parent API: preserves invariants
    } else {
        // Series ended : mark as done without rescheduling
        markAsDone();  // respects postcondition of Todo ( LSP OK )
    }
}
```

🔁 Compute the next occurrence: add period_days to the current scheduled_date (operator+ overloading will be discussed later).

- ✅ *If next ≤ end_date → schedule the next occurrence.*
- 🛑 *Else → finish the series by calling markAsDone() (from Todo).*

✨ “Extend behavior, don’t rewrite it — completeAndScheduleNext() adds recurrence logic while keeping Todo’s contract intact.”

LSP in Action — Displaying Todo/RecurringTodo Objects

07-RecurringTodo/recurring-todo.cpp

```
1  std::string recurring_todo_description(const RecurringTodo& rtodo) {
2      // Start from the base description
3      std::string text = todo_description(rtodo);
4      // Add recurrence details
5      text += " - every " + std::to_string(rtodo.periodDays()) + " days";
6      text += " until " + to_string(rtodo.endDate());
7      // If the series is finished, make it explicit
8      if (rtodo.isDone()) {
9          text += " [series complete]";
10     }
11     return text;
12 }
```

🧩 RecurringTodo inherits from Todo and preserves invariants (valid task data), preconditions (same expected inputs), postconditions (same meaning for methods).

🔄 `todo_description()` accepts any object that behaves like a Todo.

✨ "LSP allows transparent reuse → a RecurringTodo can be used wherever a Todo is expected — behavior stays consistent, code stays reusable."

Todo and RecurringTodo in Action

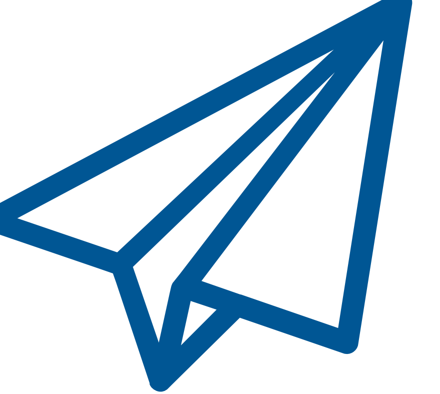
07-RecurringTodo/main.cpp

```
 1 int main() {
 2   // --- Case 1: A single Todo ---
 3   Todo todo{"Buy groceries",
 4             Date(1, 12),
 5             Category::Personal,
 6             Priority::Medium};
 7   todo.markAsDone();
 8   std::cout << todo_description(todo)
             << std::endl;
 9   // --- Case 2: A recurring Todo ---
10   RecurringTodo rent{"Run", Date(1, 5),
11                      Category::Personal,
12                      Priority::High,
13                      Date(1, 31), 3};
14   while (!rent.isDone()) {
15     std::cout << recurring_todo_description(rent)
               << std::endl;
16     rent.completeAndScheduleNext();
17   }
18   std::cout << recurring_todo_description(rent)
             << std::endl;
19   return 0;
20 }
```

```
→ 07-RecurringTodo make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c recurring-todo.cpp -o build/recurring-todo.o
clang++ -Wall -MMD -c todo.cpp -o build/todo.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/recurring-todo.o  build/todo.o  build/date.o
→ 07-RecurringTodo ./build/app
Buy groceries [Done] — Personal — Priority Medium — Due: 1/12
Run [Pending] — Personal — Priority High — Due: 1/5 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/8 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/11 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/14 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/17 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/20 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/23 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/26 — every 3 days until 1/31
Run [Pending] — Personal — Priority High — Due: 1/29 — every 3 days until 1/31
Run [Done] — Personal — Priority High — Due: 1/29 — every 3 days until 1/31 [series complete]
```

✨ "Inheritance in action: Todo defines the core logic; RecurringTodo extends it safely — respecting SRP, OCP, and LSP.."

A SIXTH TAKE HOME MESSAGE



Understanding inheritance the right way

🧩 **Inheritance lets you reuse and extend existing classes** — build on solid foundations instead of rewriting code.

🎯 **Each class keeps one clear responsibility** — extend only when the new class adds specific behavior.

🛡️ **Encapsulation matters** — keep parent attributes private and use its API to preserve invariants.

🧱 **Respect design principles** (SRP, OCP, LSP) — extend safely, don't break existing contracts.

Questions

?

Contacts

Pr. Dominique Ginhac

dginhac@ube.fr

Come visit us at

https://github.com/dginhac/polytech-dijon-itc313

This work is **licensed** under a
Creative Commons Attribution-NonCommercial 4.0 International License.

https://creativecommons.org/licenses/by-nc/4.0/