POLYTECH
DIJON

2025/2026

ITC313

Mastering C++:

🎨 The Art of Object-Oriented Programming

Pr. Dominique Ginhac
dginhac@ube.fr

Last updated on Nov, 26, 2025 by dG

https://ginhac.com/ITC313/03-polymorphism.pdf

Lecture #02

https://ginhac.com/ITC313/03-polymorphism.pdf

All code samples are available on GitHub in directory "samples/03-polymorphism"

Polymorphism

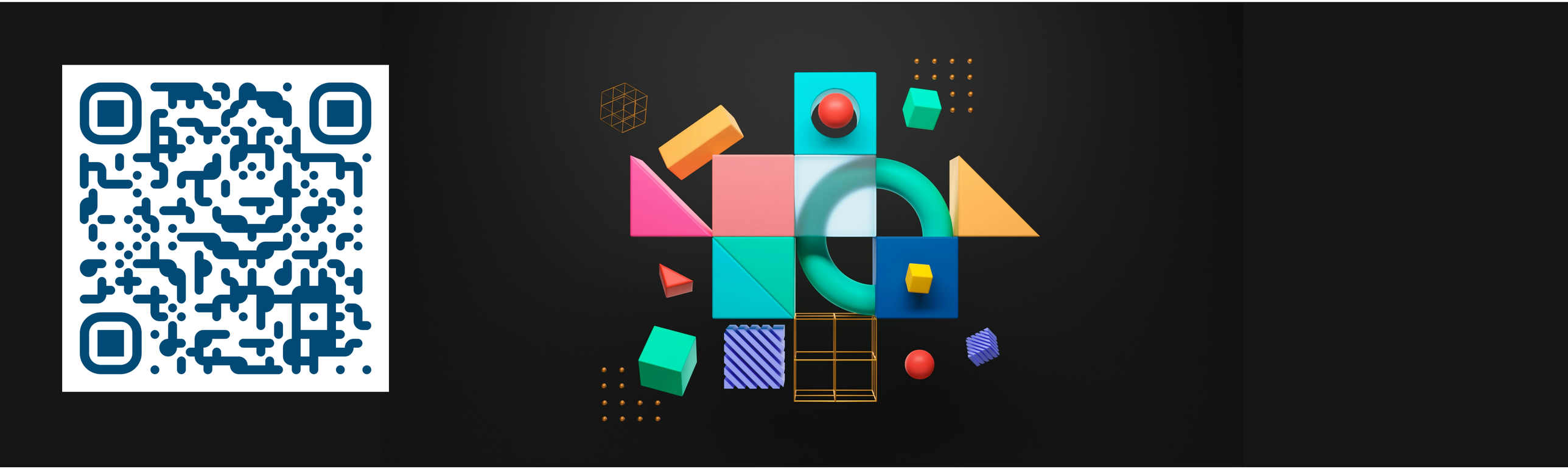


Photo by Vimal S on Unsplash

From shared structures to adaptable behaviors

Exploring polymorphism — how objects respond differently to the same message.

Lecture #00
Course Introduction

Lecture #01
User-defined Data Types
Abstraction/Encapsulation

Lecture #02
Inheritance

Lecture #03
Polymorphism

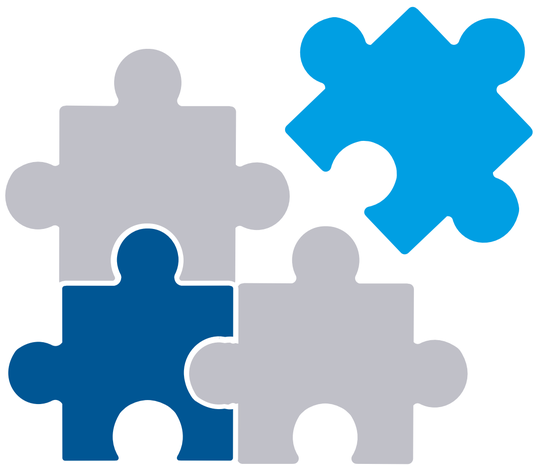
Today

Lecture #04
STL Containers

Lecture #05
Templates

...

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.

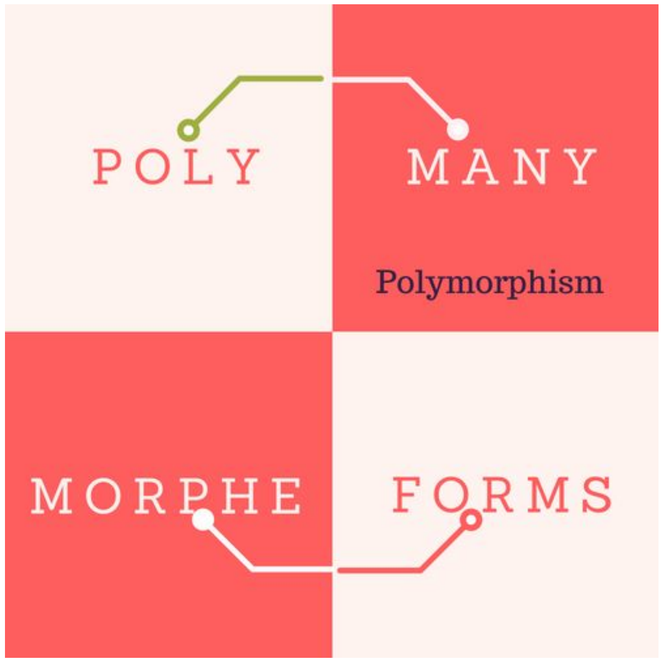
Ex: an Ebook inherits from a Book, reusing attributes like title and author and adding specific features like url or download().

🔀 **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 Polymorphism Means?

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



💡 Etymology

The word Polymorphism comes from the Greek roots poly (πολύ) = "many" and morphē (μορφή) = "form".

🧠 Definition

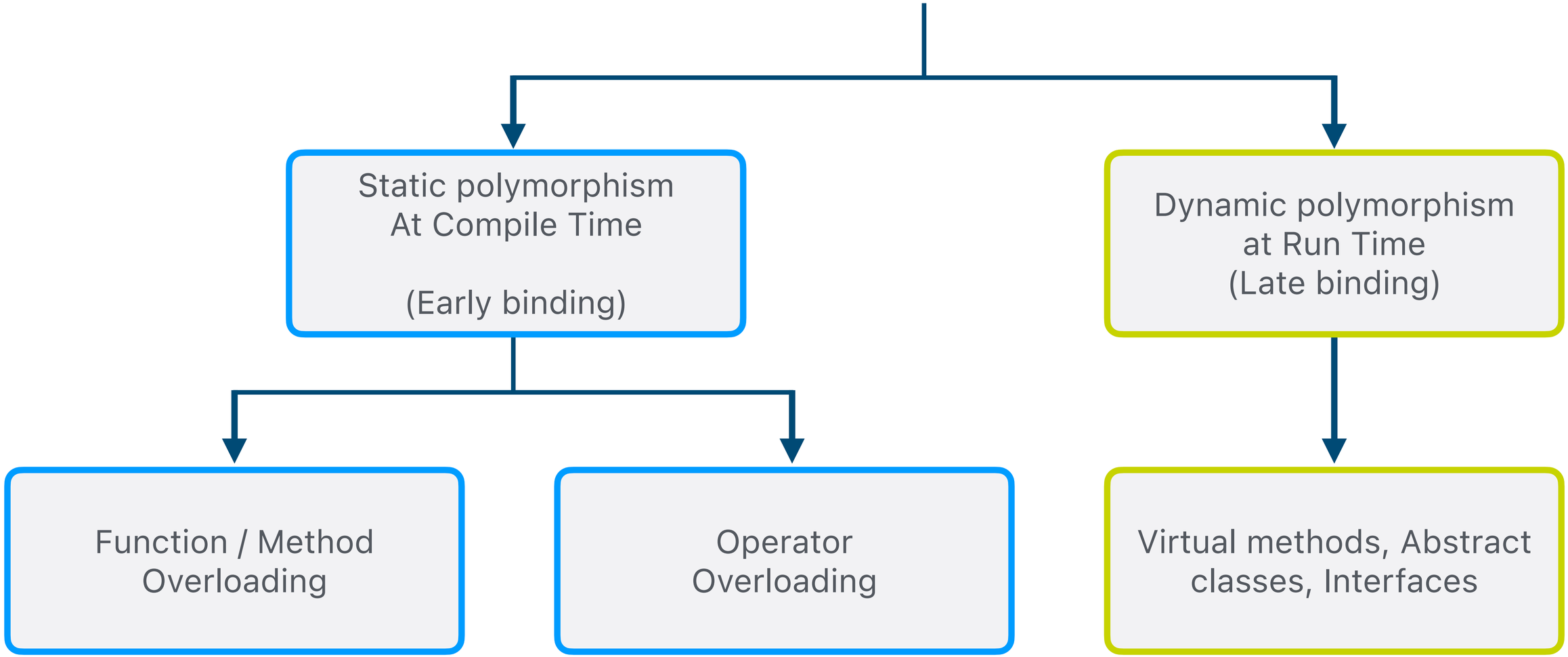
→ Polymorphism means "having many forms."

⚙️ In C++

Polymorphism allows an object to behave differently depending on the context or the type it is accessed through.

✨ "One message — many possible forms of behavior."

Two Kinds of Polymorphism



✨ “Static: the compiler decides. Dynamic: the object decides.”

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

AGENDA

03 - Polymorphism

1. Functions/methods ov
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

Function/Method Overloading: Same Name, Different Shapes

💡Function/Method overloading means defining multiple functions/methods with the same name but different parameter lists (number or types).

✅

```
void my_function(int a);
int my_function(float a);
double my_function(int a, double b);
double my_function(double a, int b);

void MyClass::myMethod(int a);
int MyClass::myMethod(float a);
double MyClass::myMethod(int a, double b);
double MyClass::myMethod(double a, int b);
```

❌

```
int my_function(int a);
double my_function(int a);
int MyClass::myMethod(float a);
double MyClass::myMethod(float a);
```

💡 Same name, different parameter list.

🧩 Works for both functions and methods.

⚙️ The compiler chooses the right version at compile time.

🛑 Compilation error!

🔎 The return type is NOT considered when resolving overloads.

⚠️ Functions that differ only by return type cause ambiguity.

✨ “Return type ≠ overload key.”

Function Overloading Example: Absolute()

💡We want to create a function that computes the absolute value of a number. It should work for both integers (int) and floating-point values (float/double).

01-function-overloading/arithmetic.cpp

```cpp
int absolute(int value) {
    // Standard if
    if (value < 0)
        value = -value;
    return value;
}
float absolute(float value) {
    // Immediate if - Conditional operator
    return value < 0 ?  -value : value;
}
double absolute(double value) {
    return value < 0 ?  -value : value;
}
```

01-function-overloading/main.cpp

```cpp
int main() {
    int a = -5;
    float b = 5.5;
    std::cout << "Abs(" << a << ") = "
              << absolute(a) << std::endl;
    std::cout << "Abs(" << b << ") = "
              << absolute(b) << std::endl;
    // Directly using a literal
    // -> calls the double version
    std::cout << “Abs(3.14) = "
              << absolute(3.14) << std::endl;
    return 0;
}
```

```
➜  01-function-overloading make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c arithmetic.cpp -o build/arithmetic.o
clang++ -Wall  -o build/app  build/main.o  build/arithmetic.o
➜  01-function-overloading ./build/app
Abs(-5) = 5
Abs(5.5) = 5.5
Abs(3.14) = 3.14
```

✨ “Overloading = one intention, multiple implementations.”

Method Overloading Example: Download()

💡In a digital world, an Ebook can be downloaded in different ways. We use method overloading to adapt the same action to different situations — full book or selected pages.

02-method-overloading/ebook.h

```
class Ebook : public Book {
private:
    std::string url_;
public:
    Ebook (const std::string& title,
           const Author& author,
           const std::string& url);
    void download() const;
    void download(int page) const;
    void download(int start, int end) const;
    std::string description () const;
};
```

02-method-overloading/main.cpp

```
int main() {
    Author author("George Orwell");
    Ebook ebook("1984", author,
                "http://example.com/1984");
    ebook.download();
    ebook.download(42);
    ebook.download(10, 20);
    return 0;
}
```

```
→  02-method-overloading 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

→  02-method-overloading ./build/app
Downloading the full book...
Downloading page 42...
Downloading from page 10 to page 20...
```

✨ “One action — multiple ways to perform it.”

Questions

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

Operator Overloading: Making Classes Feel Natural

💡 Concept

In C++, operator overloading lets us give custom meaning to standard operators (+, /, ==, <<, ...) when they are applied to user-defined classes.

🧠 Goal

To let programmers write expressions naturally, using the same syntax for objects as for primitive types.

📦 Example

```
// Without operator overloading
calculation = add(divide(a, b), multiply(a, b));

// With operator overloading
calculation = (a / b) + (a * b);
```

✅ Benefits

Keeps code consistent with mathematical intuition.

✨ “Operator overloading turns code into a language humans already speak.”

Operator Overloading: Some Real Examples

💬 Idea

Operator overloading helps us think in the problem domain, not in the machine's syntax.

📅 Example: Making Date objects intuitive

🧱 With integers	📆 With Dates
`2 + 3 = 5`	`Date(5,26) + 3 → Date(5,29)`
`2 < 3`	`Date(5,26) < Date(5,29)`

✅ Benefits

More expressive and readable code.

✨ "Operator overloading lets objects behave like numbers — clear, elegant, human."

Overloading in Action: Comparison of Date

🧠 To compare 📅 Date objects naturally, we can redefine standard operators like <, >, or ==.

💬 Operator overloading is done by defining a **special method** whose name begins with the **keyword operator** followed by the symbol to redefine.

```
ReturnType Class::operator<symbol>(ParameterList);
```

03-Date/date.h

```h
class Date {
public:
   Date(int month=1, int day=1);
   int month() const;
   int day() const;
   bool operator < (const Date& other) const;
private:
   int month_;
   int day_;
};
```

03-Date/date.cpp

```cpp
bool Date::operator < (const Date& other) const {
    if (month_<other.month_)
        return true;
    if ((month_ == other.month_) &&
        (day_<other.day_))
        return true;
    return false;
}
```

Operator < vs. Method Is_Before()

💡Operator overloading doesn't invent new behavior — it's just **a different syntax** for an existing logic. It replaces standard methods by more intuitive and readable code.

03-Date/date.cpp

```cpp
// Classic method
bool Date::isBefore(const Date& other) const {
    if (month_<other.month_) return true;
    if ((month_ == other.month_) && (day_<other.day_)) return true;
    return false;
}

// Overloaded operator<
bool Date::operator < (const Date& other) const {
    if (month_<other.month_) return true;
    if ((month_ == other.month_) && (day_<other.day_)) return true;
    return false;
}
```

Operator < vs. Method Is_Before()

03-Date/main.cpp

```cpp
int main() {
    Date a_day(5, 26);
    Date another_day(10,31);
    // Using the member function is_before
    if (a_day.isBefore(another_day)) {
        std::cout << to_string(a_day) << " is before " << to_string(another_day) << std::endl;
    } else {
        std::cout << to_string(a_day) << " is not before " << to_string(another_day)  << std::endl;
    }
    // Using the overloaded operator <
    if (a_day < another_day) {
        std::cout << to_string(a_day) << " is before " << to_string(another_day) << std::endl;
    } else {
        std::cout << to_string(a_day) << " is not before " << to_string(another_day)  << std::endl;
    }
    return 0;
}
```

```
→  03-Date make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
```

```
→  03-Date ./build/app
5/26 is before 10/31
5/26 is before 10/31
```

Operator < vs. Method Is_Before()

💡Operator overloading doesn't add new logic — it only provides a **more natural syntax** for behavior that could already be written as a normal method.

🧠 The expression `d1 < d2` is just **a shortcut** for `d1.operator<(d2)` — the same method call, written in a **human-friendly** way. Similar to the call of a standard method such as `a_day.is_before(another_day);`

03-Date/main.cpp

```cpp
int main() {
    Date a_day(5, 26);
    Date another_day(10,31);
    // Using the non human-friendly operator<
    if (a_day.operator<(another_day)) {
        std::cout << to_string(a_day) << " is before " << to_string(another_day) << std::endl;
    } else {
        std::cout << to_string(a_day) << " is not before " << to_string(another_day)  << std::endl;
    }
    return 0;
}
```

```
→  03-Date make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o

→  03-Date ./build/app
5/26 is before 10/31
```

✨ "Operator overloading is syntactic sugar — same method, sweeter syntax"

Building a Comparison Interface for Date

04-Date-comparison-operators/date.h

```
Class Date {
public:
    bool operator == (const Date& other) const; // d1 == d2
    bool operator != (const Date& other) const; // d1 != d2
    bool operator < (const Date& other) const;  // d1 <  d2
    bool operator > (const Date& other) const;  // d1 >  d2
    bool operator <= (const Date& other) const; // d1 <= d2
    bool operator >= (const Date& other) const; // d1 <= d2
private:
    int month_;
    int day_;
};
```

🔎 Key points

bool → the result of a comparison is always true or false.

const Date& other → the second operand of the comparison.

const at the end → comparisons do not modify the objects.

Together, these operators make Date behave like a fully comparable type.

Implement Once, Reuse Logic

04-Date-comparison-operators/date.cpp

```cpp
bool Date::operator == (const Date& other) const {
    return (month_ == other.month_ && day_ == other.day_);
}
bool Date::operator !=(const Date& other) const {
    return !(*this == other);
}
bool Date::operator < (const Date& other) const {
    if (month_<other.month_) return true;
    if ((month_ == other.month_) &&
        (day()<other.day_)) return true;
    return false;
}
bool Date::operator>=(const Date& other) const {
    return !(*this < other); // negation of <
}
bool Date::operator > (const Date& other) const {
    // reuse operator < with swapped operands
    return other < *this;
}
bool Date::operator<=(const Date& other) const {
    // negation of < with swapped operands
    return !(other < *this);
}
```

🔎 Key points

*Inside comparison operators, choose one style — direct attributes or getters — and keep the **implementation simple and consistent**.*

*Implement only **two primitives**: operator== and operator<.*

*Define != as the **logical negation** of ==; >= as the **logical negation** of <. and > as the **converse** of < to minimize code duplication.*

"`this`" is an implicit pointer to the current object and `*this` is the current object.

✨ "Define the minimal core behavior once, let the other operators reuse it."

Using Date Comparisons

04-Date/main.cpp

```cpp
int main() {
    Date today(5, 26);
    Date start(5, 1);
    Date deadline(5, 31);

    if (today < deadline) {
        std::cout << "There is still time before the deadline." << std::endl;
    }

    if (today >= start && today <= deadline) {
        std::cout << "Today is inside the valid period." << std::endl;
    }
    return 0;
}
```

```
→  04-Date-comparison-operators make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
→  04-Date-comparison-operators ./build/app
There is still time before the deadline.
Today is inside the valid period.
```

✨ “The code reads almost like **natural language** and Client code **does not care** about how comparison is implemented.”

Arithmetic Operators For Date

05-Date-arithmetic-operators/date.h

```
1  class Date {
2  public:
3     Date operator + (int days) const; // date + integer
4     Date operator - (int days) const; // date - integer
5     Date& operator += (int days);     // date += integer
6     Date& operator -= (int days);     // date -= integer
7  private:
8     int month_;
9     int day_;
10 };
```

🆕 `Date ± int` return a **new Date**

`d+n;` and `d-n;` never modify the original object and return a fresh Date, following the semantics of primitive arithmetic.

✏️ `Date ±= int` modify the **current Date in place**

`d+=n;` and `d-=n;` directly update d.

More efficient and concise than `d=d+n;` Behaves exactly like integer types (`i+=3;`)

🔒 **const** applies only to **±**

operator+ and operator- end with const → they guarantee immutability.

*operator+= and operator-= cannot be const because they modify *this.*

Implementing Date::Operator+

05-Date-arithmetic-operators/date.cpp

```cpp
Date Date::operator+(int days) const {
    if (days < 0) {                 // if days < 0, call Date - (-days)
        return *this - (-days);
    }
    int new_day = day_ + days;  // the new day is ok
                                   if new_day < end of month
    int new_month = month_;     // the new month starts as
                                   the current month
    int days_in_month = get_days_in_month(new_month);

    // increment month while new_day exceeds days in month
    while (new_day > days_in_month) {
        new_day -= days_in_month;   // shift into next month
        ++new_month;
        if (new_month > 12) {
            new_month = 1;          // wrap around the year
                                       (no year field here)
        }
        days_in_month = get_days_in_month(new_month);
    }
    return Date(new_month, new_day);
}
```

🔎 Key points

Handles negative input via operator-: *a call like d + (-10) is redirected to d - 10, keeping the logic clean and consistent.*

Adds days one month at a time: *if the total exceeds the number of days in the month, we wrap to the next month and continue subtracting months.*

Returns a new Date object: *operator+ does not modify the original date, it creates and returns a new corrected date.*

✨ "`Date + int` moves forward in time while respecting month boundaries."

Implementing Date::Operator-

05-Date-arithmetic-operators/date.cpp

```cpp
Date Date::operator-(int days) const {
    if (days < 0) {             //if days <0, we call Date + (-days)
        return *this + (-days);
    }
    int new_day   = day_ - days;  // the new day is ok if > 0
    int new_month = month_;

    // We move backwards month by month as long as the day is <= 0
    while (new_day <= 0) {
        --new_month;
        if (new_month < 1) {      // wrap around:
                                     before January -> December
            new_month = 12;
        }
        int days_in_month = get_days_in_month(new_month);
        new_day += days_in_month;  // we add a full month of days
    }
    return Date(new_month, new_day);
}
```

🔎 Key points

operator- is fully symmetrical to operator+: *It moves backward in time using the same month-by-month logic.*

Negative values reuse the logic of operator+: *d - (-n) automatically becomes d + n, mirroring the behavior of operator+.*

Add days from previous months: *when subtracting days pushes the date before day 1, we wrap to the previous month.*

✨ "`Date - int` mirrors the logic of `Date + int`, just in the opposite direction."."

Implementing Date::Operator±=

🔁 Reuse existing logic

Both operators rely on operator+ / operator-.

No duplication of month-boundary logic.

Guarantee of consistent behavior across all date operations.

✏️ In-place modification

*+= and -= modify *this directly.*

More expressive and more efficient than d = d + days;

🔒 Reference return type

`return *this;` enables chaining: `(d+=2)+=5;`

05-Date-arithmetic-operators/date.cpp

```cpp
Date& Date::operator+=(int days) {
    *this = *this + days; // reuse operator+
    return *this;
}

Date& Date::operator-=(int days) {
    *this = *this - days; // reuse operator-
    return *this;
}
```

✨ “+= and -= do not reimplement specific logic. They simply reuse operator+ and operator- and apply the result to *this."

Use Cases for ± and ±=

05-Date-arithmetic-operators/main.cpp

```cpp
int main() {
    Date d1(1, 28);      // January 28
    Date d2(3, 5);       // March 5
    Date result1 = d1 + 5;    // February 2
    Date result2 = d2 - 10;   // February 24
    Date future  = d1 + 40;   // March 9
    Date past    = d2 - 100;  // November 25
    d1 += 5;                  // February 2
    d2 -= 10;                 // February 23
    std::cout << "d1" << " = " << to_string(d1) << std::endl;
    std::cout << "d2" << " = " << to_string(d2) << std::endl;
    std::cout << "d1 + 5  = " <<  to_string(result1) << std::endl;
    std::cout << "d2 - 10 = " <<  to_string(result2) << std::endl;
    std::cout << "d1 + 40 = " <<  to_string(future) << std::endl;
    std::cout << "d2 - 100 = " <<  to_string(past) << std::endl;
    std::cout << "d1 += 5 = " <<  to_string(d1) << std::endl;
    std::cout << "d2 -= 10 = " << to_string(d2) << std::endl;
    return 0;
}
```

```
→  05-Date-arithmetic-operators make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o

→  05-Date-arithmetic-operators ./build/app
d1 = 2/2
d2 = 2/23
d1 + 5  = 2/2
d2 - 10 = 2/23
d1 + 40 = 3/9
d2 - 100 = 11/25
d1 += 5 = 2/2
d2 -= 10 = 2/23
```

✨ “Arithmetic on dates remains expressive and readable, even when spanning multiple months.”

Before Applying ++ / -- to Date...

Before applying prefix and postfix operators to our Date class, we first examine their behavior on integers. The rules are the same — but much easier to see on simple types.

06-Int-inc-dec—operators/main.cpp

```cpp
int main() {
    int i = 5;
    i++; // postfix operator - returns an rvalue
    std::cout << "i after i++: " << i << std::endl; // Outputs 6
    ++i; // prefix operator -- returns an lvalue
    std::cout << "i after ++i: " << i << std::endl; // Outputs 7
    ++++i; // RIGHT: equivalent to ++(++i) because prefix returns an lvalue
    // i++++; ERROR: postfix does not return an lvalue
    std::cout << "i after ++++i: " << i << std::endl; // Outputs 9
    (++i)++; // RIGHT: postfix applied to the lvalue returned by prefix
    // ++(i++) or ++i++; ERROR: prefix cannot be applied to the rvalue returned by
    std::cout << "i after (++i)++: " << i << std::endl; // Outputs 11
    int j = ++i;   // i becomes 12, j = 12
    std::cout << "i,j after j = ++i: " << i << "," << j << std::endl; // Outputs 12,12
    j = i++; // j = 12, then i becomes 13
    std::cout << "i,j after j = i++: " << i << "," << j << std::endl; // Outputs 13,12
    return 0;
}
```

```
→  06-Int-inc-dec-operators make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall  -o build/app  build/main.o

→  06-Int-inc-dec-operators ./build/app
i after i++: 6
i after ++i: 7
i after ++++i: 9
i after (++i)++: 11
i,j after j = ++i: 12,12
i,j after j = i++: 13,12
```

Increment (++) and Decrement (--) Operators for Date

07-Date-inc-dec—operators/date.h

```
class Date {
public:
   Date& operator ++ (); // prefix increment: ++date
   Date& operator -- (); // prefix decrement: --date
   Date operator ++ (int); // use a dummy int param for postfix increment: date++
   Date operator -- (int); // use a  dummy int param for postfix decrement: date--
private:
   int month_;
   int day_;
};
```

🔎 Key points

***Four operators**: prefix (++d, --d) and postfix (d++, d--) are all overloaded.*

*C++ separates them using a **dummy int parameter in postfix**: this parameter is never used; it exists only to differentiate signatures.*

***Prefix returns a reference**: it returns the same object, already modified (lvalue).*

***Postfix returns a value**: it returns a copy of the old date (rvalue), then updates the object.*

***No const** at the end: all these operators return a new object but also modify the current Date.*

Implementing Date::Operator++/--

07-Date-inc-dec—operators/date.cpp

```cpp
Date& Date::operator ++() {
    // prefix increment
    *this += 1; // reuse Date += int
    return *this;
}
Date& Date::operator --()  {
    // prefix decrement
    *this -= 1; // reuse Date -= int
    return *this;
}
Date Date::operator ++(int) {
    // postfix increment
    Date tmp = *this;
    *this += 1; // reuse Date += int
    return tmp;
}
Date Date::operator --(int) {
    // postfix decrement
    Date tmp = *this;
    *this -= 1; // reuse Date - int
    return tmp;
}
```

🔁 ++ and -- delegate to operator+ / operator-

Reuses `Date ±= 1`, so there is only one place where Date logic lives.

📎 Prefix (`++d`, `--d`) returns a reference

Prefix updates the object and then returns the object itself, not a copy.

🔀 Assignment behavior of prefix ++/-- depends on the left-hand side

If assigned to a variable → always a copy → independent object.

If assigned to a reference → aliasing the object.

📄 Postfix (`d++`, `d--`) always returns a value

Postfix returns a temporary copy of the old value (rvalue) that cannot be modified or bound to a reference.

The update is performed after saving the old state.

First Use Case for Prefix & Postfix ++ / --

07-Date-inc-dec-operators/case1.cpp

```cpp
int main() {
    Date d1(5, 4);      // May 4
    std::cout << "init -> d1 = " << to_string(d1) << std::endl;
    d1++; // May 5
    std::cout << "d1++ -> d1 = " << to_string(d1) << std::endl;
    ++d1; // May 6
    std::cout << "++d1 -> d1 = " << to_string(d1) << std::endl;
    return 0;
}
```

```
→  07-Date-inc-dec-operators make -f Makefile1
clang++ -Wall -MMD -c case1.cpp -o build/case1.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app1  build/case1.o  build/date.o
```

```
→  07-Date-inc-dec-operators ./build/app1
init -> d1 = 5/4
d1++ -> d1 = 5/5
++d1 -> d1 = 5/6
```

✨ “As easy to use on Date objects as on basic integer types.”

Second Use Case for Prefix & Postfix ++ / --

07-Date-inc-dec-operators/case2.cpp

```cpp
int main() {
    Date d1(5, 4);      // May 4
    std::cout << "init           -> d1 = " << to_string(d1) << std::endl;
    Date d2 = ++d1;     // Prefix: d1 = d2 (new object) = May 5
    std::cout << "d2 = ++d1      -> d1 = " << to_string(d1) << " ; ";
    std::cout << "d2 = " << to_string(d2) << std::endl;
    d1 = Date(3,14);  // Update d1 to March 14, d2 unchanged = May 5
    std::cout << "Update to 3/14 -> d1 = " << to_string(d1) << " ; ";
    std::cout << "d2 = " << to_string(d2) << std::endl << std::endl;

    d1 = Date(5, 4);      // May 4
    std::cout << "Reset to 5/4  -> d1 = " << to_string(d1) << std::endl;
    Date d3 = d1++;   // Postfix: d1 = May 5, d3 (new object) = May 4
    std::cout << "d3 = d1++      -> d1 = " << to_string(d1) << " ; ";
    std::cout << "d3 = " << to_string(d3) << std::endl;
    d1 = Date(3,14);  // Update d1 to March 14, d3 unchanged = May 5
    std::cout << "Update         -> d1 = " << to_string(d1) << " ; ";
    std::cout << "d3 = " << to_string(d3) << std::endl;
    return 0;
}
```

```
→  07-Date-inc-dec-operators make -f Makefile2
clang++ -Wall -MMD -c case2.cpp -o build/case2.
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app2  build/case2.o  bu
```

```
→  07-Date-inc-dec-operators ./build/app2
init       -> d1 = 5/4
d2 = ++d1  -> d1 = 5/5 ; d2 = 5/5
d3 = d1++  -> d1 = 5/6 ; d3 = 5/5
Reset      -> d1 = 5/4 ; d3 = 5/5
```

✨ “Postfix operator (d++) → Object copy.
Prefix operator (++d) assigned to a variable → Object copy.”

Final Use Case for Prefix & Postfix ++ / --

07-Date-inc-dec-operators/case3.cpp

```cpp
int main() {
    Date d1(5, 4);      // May 4
    std::cout << "init          -> d1 = " << to_string(d1) << std::endl;
    // Date& rd1 = ++d1 -> Prefix operator assignment to reference
    // Date& rd2 = d1++ -> Postfix operator -> COMPILER ERROR: cannot bind reference to temporary
    Date& rd1 = ++d1; // d1 and rd1 = May 5
    std::cout << "rd1 = ++d1   -> d1 = " << to_string(d1) << " ; ";
    std::cout << "rd1" << " = " << to_string(rd1) << std::endl;
    d1 = Date(5,4);  // reset d1 and rd1 to May 4
    std::cout << "Reset to 5/4 -> d1 = " << to_string(d1) << " ; ";
    std::cout << "rd1 = " << to_string(rd1) << std::endl;
    return 0;
}
```

```
→  07-Date-inc-dec-operators make -f Makefile3
clang++ -Wall -MMD -c case3.cpp -o build/case3.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app3  build/case3.o  build/date.o
```

```
→  07-Date-inc-dec-operators ./build/app3
init          -> d1 = 5/4
rd1 = ++d1    -> d1 = 5/5 ; rd1 = 5/5
Reset to 5/4 -> d1 = 5/4 ; rd1 = 5/4
```

✨ “Prefix operator (++d) assigned to a reference → Object aliasing.”

Not all Operators Can Be Implemented as Methods

📦 Member operators require the **left operand** to be an **instance of the class**

→ *`date + 3;` is equivalent to `date.operator+(3);`*

↔️ But sometimes the object only appears on the **right-hand** side

→ *Example: `3 + date;`*

🚫 In this case, **a method cannot be used**

Because the left operand (3) is not a Date object.

🔧 Solution: use a **free function**

Optionally declared as friend if it needs access to private data.

> 📌 ***Pedagogical note:***
>
> *`int + Date` is used here for teaching purposes only.*
>
> *In practice, `Date + int` is the natural and recommended operation.*

✨ “Method operators work only when the object is on the left. If the object is on the right, you must use a non-member (free) operator.”

Enabling Int + Date: Free Vs Friend Function

08-Date-free-operators/date.h

```
1  class Date {
2  public:
3      // date + integer
4      Date operator + (int days) const;
5  private:
6      int month_;
7      int day_;
8  };
9  // Free function -> integer + date
10 Date operator + (int days, const Date& date);
```

09-Date-friend-operators/date.h

```
1  class Date {
2  public:
3      // Method -> date + integer
4      Date operator + (int days) const;
5      // Friend non-member function -> integer + date
6      friend Date operator + (int days, const Date& date);
7  private:
8      int month_;
9      int day_;
10 };
```

📦 Same **syntax** for the user

Both versions enable the exact same expression: `Date d2 = 3 + d1;`

From the caller's point of view, there is no difference.

🏗️ Different places of **declaration**

*Free function is declared outside the class — Friend function is declared inside the class body with the **friend** keyword.*

🔀 **Two parameters** (`int days, const Date& date`) with a specific order

The order reflects the expression `int + Date` — Left operand = `int`, Right operand = `Date`

Implementing Int + Date: Free Vs Friend Function

08-Date-free-operators/date.cpp

```cpp
Date operator + (int days, const Date& date) {
    return date + days; // reuse Date + int
}
```

09-Date-friend-operators/date.cpp

```cpp
Date operator + (int days, const Date& date) {
    return date + days; // reuse Date + int
}
```

🔁 Same **implementation** with reuse of `Date + int`

The logic of date arithmetic remains centralized — No duplication of month/day handling code.

🔒 Free vs Friend: **access level difference**

Free function does have access to private members (month_, day_) → restricted to the public interface (getters, Date + int, etc.).

Friend function has full access to Date's internals → it is "trusted" by the class.

⚙️ Same **behavior** and **performance**

When well designed, both versions compile to equivalent code.

The real difference is design and encapsulation, not speed.

✨ "Friend functions allow privileged access — but good design prefer Free functions by reusing the existing class' public API."

Testing Int + Date: Utility vs Friend Function

08-Date-free-operators/main.cpp

09-Date-friend-operators/main.cpp

```cpp
int main() {
    Date d1(5, 4);      // May 4
    std::cout << "init -> d1 = " << to_string(d1) << std::endl;
    Date d2 = d1 + 5; // May 9
    std::cout << "d2 = d1 + 5 -> d2 = " << to_string(d2) << std::endl;
    d2 = 3 + d1; // May 7
    std::cout << "d2 = 3 + d1 -> d2 = " << to_string(d2) << std::endl;
    return 0;
}
```

```
→  08-Date-free-operators make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
→  08-Date-free-operators ./build/app
init -> d1 = 5/4
d2 = d1 + 5 -> d2 = 5/9
d2 = 3 + d1 -> d2 = 5/7
```

```
→  09-Date-friend-operators make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
→  09-Date-friend-operators ./build/app
init -> d1 = 5/4
d2 = d1 + 5 -> d2 = 5/9
d2 = 3 + d1 -> d2 = 5/7
```

🔄 Implementation choice is **transparent** to the user

The calling syntax is identical.

The behavior is identical, only the internal design differs.

🔐 **Design** decisions do not leak into user code

Friend or non-friend is a class designer's decision.

The user only sees a clean and consistent interface.

Choosing Between Method, Free, and Friend

	🔶 *Properties*	📦 *Appropriate to*	⚠️ *Limitations*
Method	Declared and defined **inside** the class as a full member.	+++ The **left operand** is the object. +++ The operation is clearly part of the **core interface** of the type.	--- Only works when the **left operand** is of the class type.
Free	Declared and defined **outside** the class as a non-member.	+++ Expressions where the class is **not on the left**. ++ **Strong encapsulation** using existing public methods. Often recommended for **symmetric operators**.	-- **Less integrated** with the class even if it is part of the core interface.
Friend	Declared as friend **inside** the class. But defined **outside** as a non-member.	+++ Expressions where the class is **not on the left**. +++ Implementation requires **full access** to private members/attributes. Want to signal that the function is part of the "**trusted core**" of the type.	--- **Weakens encapsulation**: external code can touch internals. -- **Increases coupling** between the function and the class.

Design Strategy

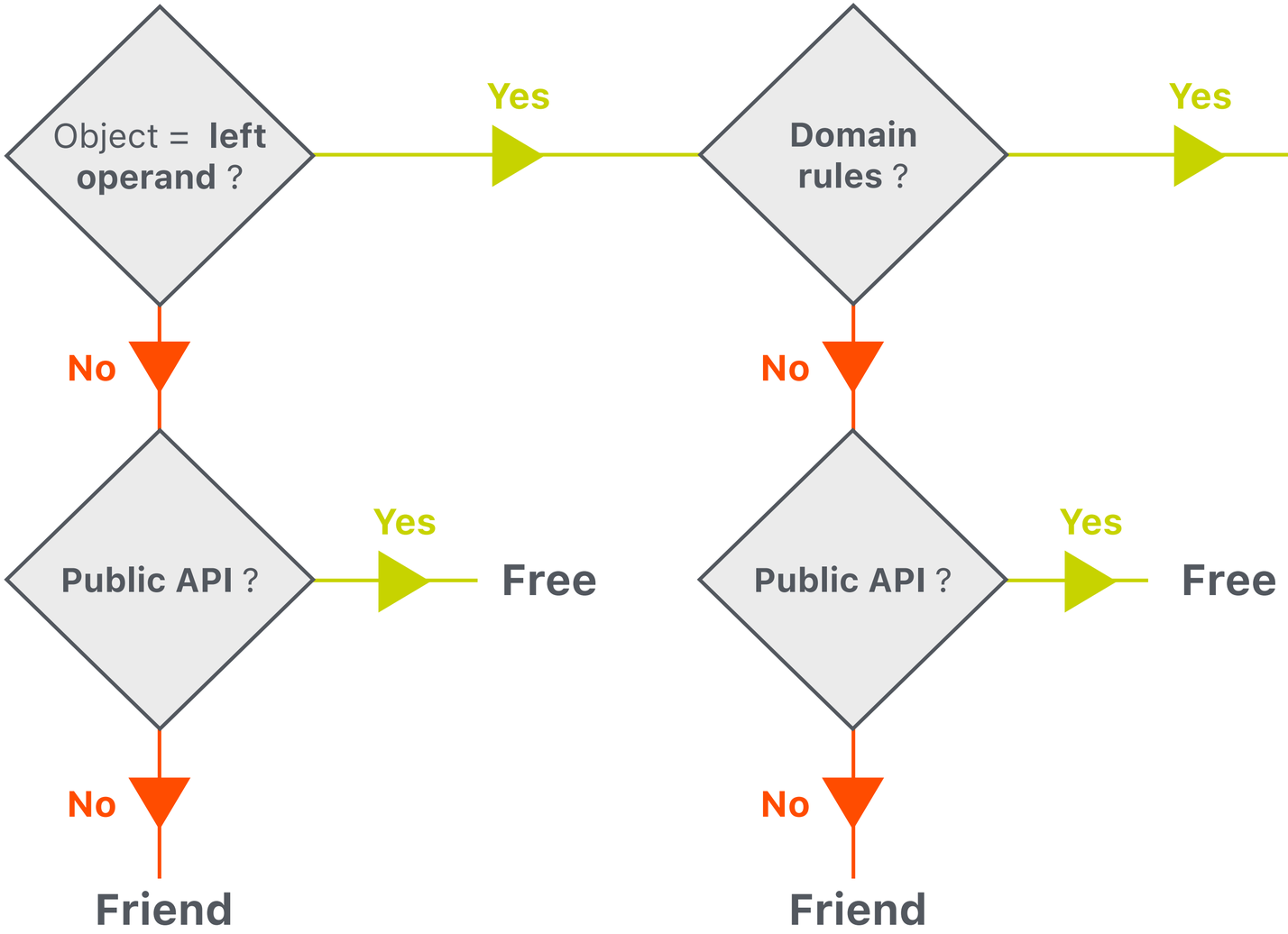


Photo by Jon Tyson on Unsplash

🏗️ 1. Start with **Method** operators

🧩 2. Add **Free functions** for symmetry

🔐 3. Use **Friend** only when necessary

Questions

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading

3.I/O overloading

4. Virtual methods and overriding
5. Abstract classes

I/O Overloading: Making Classes Behaves Like Built-in Types

💡 Concept

In C++, built-in types like int, double or std::string use the stream operators << and >> for input and output.

🧠 Goal

Overload these operators so that user-defined classes behave like built-in types.

📦 Example

```
    Date d1(5, 4);      // May 4
    // Without I/O overloading, we need to use a function to convert to string
    std::cout << << to_string(d1) << std::endl;

    // With I/O overloading, we can directly use the stream insertion operator
    std::cout << d1 << std::endl;
```

✅ Benefits

More natural syntax, consistency with built-in types, and improved code readability.

✨ “I/O overloading brings user-defined types into the standard C++ I/O world.”

Classic Approach vs Stream Operator Overloading

10-Date-IO-operators/date.h

```
1 std::string to_string(const Date& d) {
2     return std::to_string(d.month()) + “/"
            + std::to_string(d.day());
3 }
```

10-Date-IO-operators/date.h

```
1 std::ostream& operator << (std::ostream& os, const Date& d) {
2     os << std::to_string(d.month()) + "/"
          + std::to_string(d.day());
3     return os;
4 }
```

🧵 Returns a **std::string**

Provides a pure textual representation of the object.

Explicitly separates formatting from output.

🧰 **Reusable** in non-stream contexts

Useful for GUI labels, dialogs, etc.

❌ Extra step for stream output

Requires `std::cout << to_string(date);`

🔗 Must be a **free function**

The left operand is std::ostream&, not Date → operator<< is implemented as a free function with 2 parameters.

Returns a reference to the stream (std::ostream&) for chaining multiple <<.

🖨️ Becomes the **main output API** for Date

No `to_string()` needed: `cout << date;`

🌐 **Unified display interface** for all C++ streams

Works seamlessly with standard streams (std:cin, std::cout), debug stream (std::cerr), file streams (std::ifstream, std::ofstream), string streams (std::istringstream ,std::ostringstream), …

Testing I/O Overloading

10-Date-IO-operators/main.cpp

```cpp
int main() {
    Date d1(5, 4);      // May 4
    std::cout << "With to_string -> d1 = " << to_string(d1) << std::endl;
    std::cout << "with << -> d1 = " << d1 << std::endl;
    return 0;
}
```

```
→  10-Date-IO-operators make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
→  10-Date-IO-operators ./build/app
With to_string -> d1 = 5/4
with << -> d1 = 5/4
```

🧪 **Same result**, different approach

Both versions produce the exact same output string for the same Date.

🔍 **No** visible **difference** for the user

From the user's point of view, the only visible change is the syntax.

🌐 Better **integration** with operator<<

The stream version integrates naturally with std::cout, logs, files, and debug output.

Questions

AGENDA

03 - Polymorphism

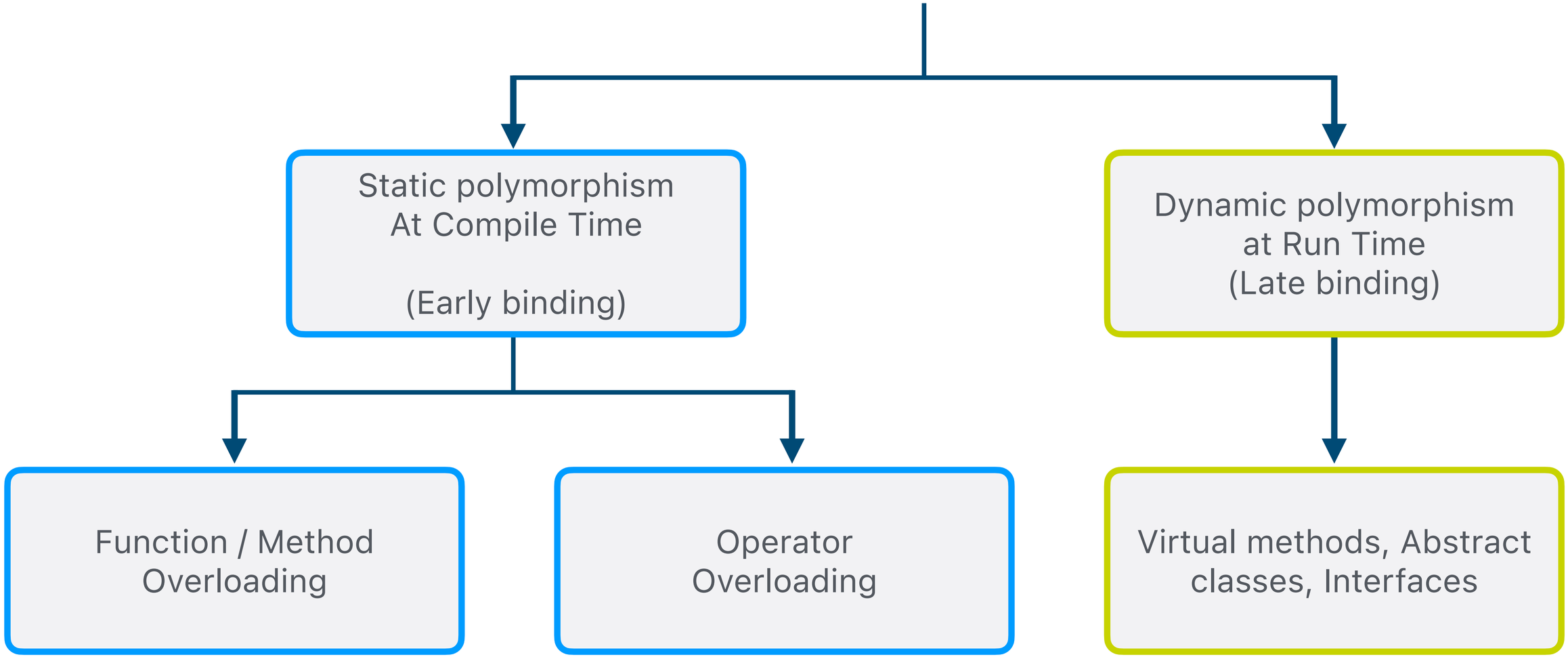
1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and c
5. Abstract classes

Two Kinds of Polymorphism



✨ “Static: the compiler decides. Dynamic: the object decides.”

Introducing isOverdue()

A simple task is overdue if its scheduled date is passed and a recurring task is overdue only when its recurrence period is over (simplified behavior for pedagogical considerations).

Same question →Different business meaning → Depending on the task nature.

11-RecurringTodo-overdue/todo.h

```
1  class Todo {
2  public:
3      bool isOverdue(const Date& date) const;
4  private:
5      std::string title_;
6      Date scheduled_date_;
7      Category category_;
8      Priority priority_;
9      bool done_;
10 };
```

11-RecurringTodo-overdue/todo.cpp

```
1  bool Todo::isOverdue(const Date& date) const {
2      return !isDone() && scheduledDate() < date;
3  }
```

11-RecurringTodo-overdue/recurring-todo.h

```
1  class RecurringTodo : public Todo {
2  public:
3      bool isOverdue(const Date& date) const;
4  private:
5      Date end_date_;
6      int period_days_;
7  };
```

11-RecurringTodo-overdue/recurring-todo.cpp

```
1  bool RecurringTodo::isOverdue(const Date& date)
const {
2      return !isDone()
3          && endDate() < date;
4  }
```

Testing isOverdue()

```
→  11-RecurringTodo-overdue make -f Makefile1
clang++ -Wall -MMD -c case1.cpp -o build/case1.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall -MMD -c todo.cpp -o build/todo.o
clang++ -Wall -MMD -c recurring-todo.cpp -o build/recurring-todo.o
clang++ -Wall  -o build/app  build/case1.o  build/date.o  build/todo.o  build/recurring-todo.o
```

```
→  11-RecurringTodo-overdue ./build/app
Buy groceries [Pending] — Personal — Priority Medium — Due: 11/11
1
Run [Pending] — Personal — Priority High — Due: 11/1 — every 3/1 days until 11/30
0
1
```

11-RecurringTodo-overdue/case1.cpp

```cpp
int main() {
    Todo groceries{"Buy groceries",
                   Date(11, 11),
                   Category::Personal,
                   Priority::Medium};
    std::cout << groceries << std::endl;
    std::cout << groceries.isOverdue(Date(12, 24)) << std::endl;

    RecurringTodo run{"Run", Date(11, 1),
                      Category::Personal,
                      Priority::High,
                      Date(11, 30), 3};
    std::cout << run << std::endl;
    std::cout << run.isOverdue(Date(11, 24)) << std::endl;
    std::cout << run.isOverdue(Date(12, 24)) << std::endl;
    return 0;
}
```

✨ “Static polymorphism: the compiler decides.”

Same Call, Wrong Logic

Imagine we define a free function `task_is_late()`.

11-RecurringTodo-overdue/todo.h

```cpp
bool task_is_late(const Todo& task, const Date& date) {
    return task.isOverdue(date);
}
```

11-RecurringTodo-overdue/case2.cpp

```cpp
int main() {
  Todo groceries{"Buy groceries", Date(11, 11),
                 Category::Personal,
                 Priority::Medium};
  RecurringTodo run{"Run", Date(11, 1),
                    Category::Personal,
                    Priority::High,
                    Date(11, 30), 3};
  std::cout << groceries.isOverdue(Date(11, 24)) << std::endl;
  std::cout << task_is_late(groceries, Date(11, 24)) << std::endl;
  std::cout << run.isOverdue(Date(11, 24)) << std::endl;
  std::cout << task_is_late(run, Date(11, 24)) << std::endl;
  return 0;
}
```

```
→  11-RecurringTodo-overdue make -f Makefile2
clang++ -Wall -MMD -c case2.cpp -o build/case2.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall -MMD -c todo.cpp -o build/todo.o
clang++ -Wall -MMD -c recurring-todo.cpp -o buil
d/recurring-todo.o
clang++ -Wall  -o build/app  build/case2.o  buil
d/date.o  build/todo.o  build/recurring-todo.o
→  11-RecurringTodo-overdue ./build/app
1
1
0
1
```

🧠 Observed behavior

⚠️ *The `Todo::isOverdue()` method is always called based on the **task's static type** used in `task_is_late()`.*

🧠 *The result is technically correct but semantically wrong.*

🚫 ***LSP violation**: a RecurringTodo is no longer a valid substitute for a Todo.*

⚙️ ***Static polymorphism**: the compiler binds the call using the static type (Todo reference).*

Making isOverdue() Virtual

💡 Core idea

*Enable **runtime polymorphism** so the right isOverdue() is called for each task type.*

⚙ Polymorphism Mechanics

💡 *Mark isOverdue() as **virtual** in Todo — virtual tells the compiler: "this method supports dynamic dispatch".*

🎯 *Mark isOverdue() as **override** in RecurringTodo to show explicit redefinition — override tells the programmer's intent: "this method redefines a virtual one".*

🗃️ *Write **virtual** and **override** only in the **header** (class declaration in .h), not in the implementation (in .cpp).*

⚙️ *virtual is required for **dynamic dispatch**. override is not required, but it makes **your intent explicit** and **the code more readable.***

🧱 *The **implementation** of isOverdue() in both classes remains **unchanged**.*

12-RecurringTodo-virtual/todo.h

```
1 class Todo {
2 public:
3   virtual bool isOverdue(const Date& date) const;
4 };
```

12-RecurringTodo-virtual/recurring-todo.h

```
1 class RecurringTodo : public Todo {
2 public:
3   bool isOverdue(const Date& date) const override;
4 };
```

✨ "virtual and override don't change your code — they change how the compiler understands it."

Testing isOverdue()

12-RecurringTodo-virtual/main.cpp

```cpp
int main() {
  Todo groceries{"Buy groceries", Date(11, 11),
                 Category::Personal,
                 Priority::Medium};
  RecurringTodo run{"Run", Date(11, 1),
                    Category::Personal,
                    Priority::High,
                    Date(11, 30), 3};
  std::cout << groceries.isOverdue(Date(11, 24))
            << std::endl;
  std::cout << task_is_late(groceries, Date(11, 24))
            << std::endl;
  std::cout << run.isOverdue(Date(11, 24))
            << std::endl;
  std::cout << task_is_late(run, Date(11, 24))
            << std::endl;
  return 0;
}
```

```
→  12-RecurringTodo-virtual make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall -MMD -c todo.cpp -o build/todo.o
clang++ -Wall -MMD -c recurring-todo.cpp -o build
/recurring-todo.o
clang++ -Wall  -o build/app  build/main.o  build/
date.o  build/todo.o  build/recurring-todo.o
```

```
→  12-RecurringTodo-virtual ./build/app
1
1
0
0
```

🧠 Observed behavior

✅ *The correct isOverdue() method is now called based on the **task's dynamic type** used in `task_is_late()`.*

🧠 *The result is technically correct and semantically consistent.*

🧩 *LSP respected: a RecurringTodo is a valid substitute for a Todo.*

⚙️ *Dynamic polymorphism: the call is bound at runtime using the real object type.*

✨ "Dynamic polymorphism: the object decides."

Questions

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding
5. Abstract classes

AGENDA

03 - Polymorphism

1. Functions/methods overloading
2. Operator overloading
3. I/O overloading
4. Virtual methods and overriding

5.Abstract classes

Abstract Classes — From Behavior to Concept

💡 Concept

*Some classes describe **concepts**, not **concrete objects**.*

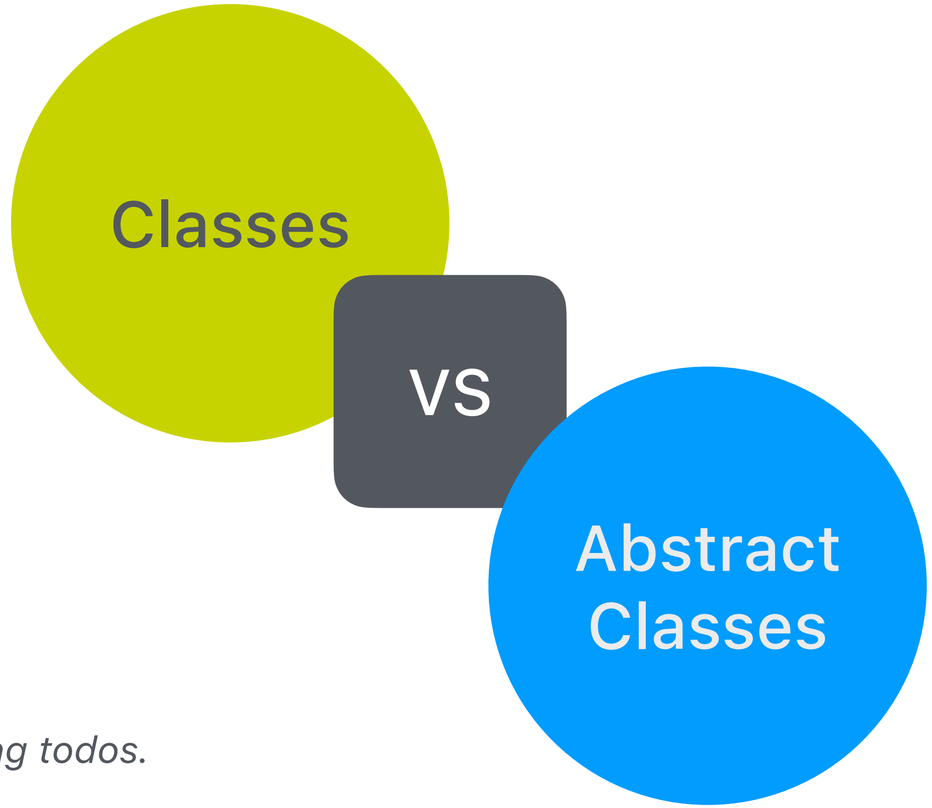
*Abstract classes provide a foundation for **reusable** and **extensible** designs.*

🧩 Abstraction Principle

- 🧠 *A base class can represent a **common concept**.*
- 🚫 *It should **not** always be directly **instantiable**.*
- 🎯 *It defines a **contract** for derived classes.*
- ⚙️ *This is done using **pure virtual functions**.*

🔀 From theory to practice

- 🧠 *Todo represents a **concept**, not a concrete task.*
- 🎯 *Let's make Todo an **abstract** class and derive concrete basic todos and recurring todos.*



✨ "Abstract classes define what objects must do, not how they do it."

Abstract Todo — a Concept, Not an Object

13-Todo-abstract/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     // Pure virtual method makes the class abstract
15     virtual bool isOverdue(const Date& date) const = 0;
16 };
```

💡 Core idea

Todo represents the concept of a task, not a concrete implementation.

🧩 What changes

- 🚫 *You can **no longer instantiate Todo** directly.*
- ⚙️ *isOverdue becomes a **pure virtual method**.*

🧱 What does not change

- 🧠 *All **domain data and attributes** remain the same.*
- 📦 *Todo still contains title, date, category, priority, status.*
- ⚙️ *Existing **getters** and **setters** are unchanged.*

✨ “An abstract class changes the role of a class, not its data or logic.”

SimpleTodo — From Concept To Object

SimpleTodo is the first concrete implementation of our abstract Todo concept.

13-Todo-abstract/simple-todo.h

```h
class SimpleTodo : public Todo {
public:
    using Todo::Todo;
    bool isOverdue(const Date& today) const;
};
```

13-Todo-abstract/simple-todo.cpp

```cpp
bool SimpleTodo::isOverdue(const Date& today) const {
    return !isDone() && scheduledDate() < today;
}
```

🧠 *SimpleTodo **inherits** from the abstract base class Todo.*

🎯 *It provides a **concrete implementation** of the pure virtual method isOverdue()*

⚙️ *`using Todo::Todo;` reuses the **base class constructor**.*

✅ *This avoids **code duplication** and keeps constructors consistent.*

⚙️ *The **inherited logic** from Todo remains untouched.*

💡 *Only the **specific behavior** of SimpleTodo is implemented here.*

🧠 *The isOverdue() method contains the business logic (`isDone()` and `scheduleDate()`) of a simple task.*

✅ *This keeps the class focused on its **own responsibility**.*

✨ “Abstract classes define rules. Concrete classes make them real.”

RecurringTodo — a Specialized Behavior

RecurringTodo extends the Todo concept with time-based recurrence behavior.

13-Todo-abstract/recurring-todo.h

```h
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);
    const Date& endDate() const;
    int periodDays() const;
    void updateEndDate(const Date& end_date);
    void updatePeriodDays(int period_days);
    void completeAndScheduleNext();
    bool isOverdue(const Date& date) const override;
private:
    Date end_date_;
    int period_days_;
};
```

🧩 Class Declaration (Header)

🧠 *RecurringTodo **inherits** from the abstract base class Todo.*

🧩 *It adds **recurrence-specific attributes**: period and end date.*

🎯 *It overrides the **pure virtual method** isOverdue().*

✅ *The class remains focused on its **own responsibility**: recurrence logic.*

🪄 Constructor design

🧠 *The constructor **extends** the base Todo constructor.*

⚙️ *It initializes both **common task data** and **recurrence-specific data**.*

RecurringTodo — a Specialized Behavior

RecurringTodo extends the Todo concept with time-based recurrence behavior.

13-Todo-abstract/recurring-todo.cpp

```cpp
void RecurringTodo::completeAndScheduleNext() {
    const Date next = scheduledDate()+ period_days_;
    if (next <= end_date_) {
        updateScheduledDate(next);
    } else {
        markAsDone();
    }
}
bool RecurringTodo::isOverdue(const Date& date)  const  {
    return !isDone()
        && endDate() < date;
}
```

💡 *Only the **recurrence-specific** behavior is implemented.*

🧠 *The **core Todo** logic remains **untouched**.*

🎯 *isOverdue() now uses `endDate()` and recurrence rules.*

✅ *This preserves **SRP**: RecurringTodo handles recurrence, not general task logic.*

SimpleTodo & RecurringTodo — Runtime Behavior Test

Now that our design is in place, let’s observe the runtime behavior of both implementations.

13-Todo-abstract/main.cpp

```cpp
int main() {
    SimpleTodo groceries{"Buy groceries", Date(11, 11),
                  Category::Personal,
                  Priority::Medium};
    RecurringTodo run{"Run", Date(11, 1),
                      Category::Personal,
                      Priority::High,
                      Date(11, 20), 3};
    std::cout << groceries << std::endl;

    while (!run.isDone()) {
            std::cout << run << std::endl;
            run.completeAndScheduleNext();
        }
    std::cout << run << std::endl;
    return 0;
}
```

```
➜  13-Todo-abstract make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall -MMD -c todo.cpp -o build/todo.o
clang++ -Wall -MMD -c simple-todo.cpp -o build/simple-todo.o
clang++ -Wall -MMD -c recurring-todo.cpp -o build/recurring-todo.o
clang++ -Wall  -o build/app  build/main.o  build/date.o  build/todo.o  build/simple-todo.o  build/rec

➜  13-Todo-abstract ./build/app
Buy groceries [Pending] — Personal — Priority Medium — Due: 11/11
Run [Pending] — Personal — Priority High — Due: 11/1 — every 3/1 days until 11/20
Run [Pending] — Personal — Priority High — Due: 11/4 — every 3/1 days until 11/20
Run [Pending] — Personal — Priority High — Due: 11/7 — every 3/1 days until 11/20
Run [Pending] — Personal — Priority High — Due: 11/10 — every 3/1 days until 11/20
Run [Pending] — Personal — Priority High — Due: 11/13 — every 3/1 days until 11/20
Run [Pending] — Personal — Priority High — Due: 11/16 — every 3/1 days until 11/20
Run [Pending] — Personal — Priority High — Due: 11/19 — every 3/1 days until 11/20
Run [Done] — Personal — Priority High — Due: 11/19 — every 3/1 days until 11/20 [series complete]
```

✨ “Concrete implementations give life to abstract concepts.”

Sorry!

More Content Coming Soon...

Contacts

Pr. Dominique Ginhac

dginhac@ube.fr

Come visit us at

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