2024/2025

ITC313

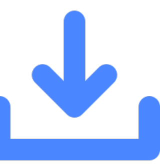
Mastering Object-Oriented Programming in C++: From Fundamentals to Best Practices

Pr. Dominique Ginhac
dginhac@u-bourgogne.fr

Last updated on November, 6, 2024 by dG

Lecture #03

Polymorphism



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

Photo by Braňo on Unsplash

Introduce polymorphism one of the most important pillars of OOP with concrete examples

Enjoy!

Lecture #00
Course Introduction

Lecture #01
User-defined Data Types

Lecture #02
Inheritance

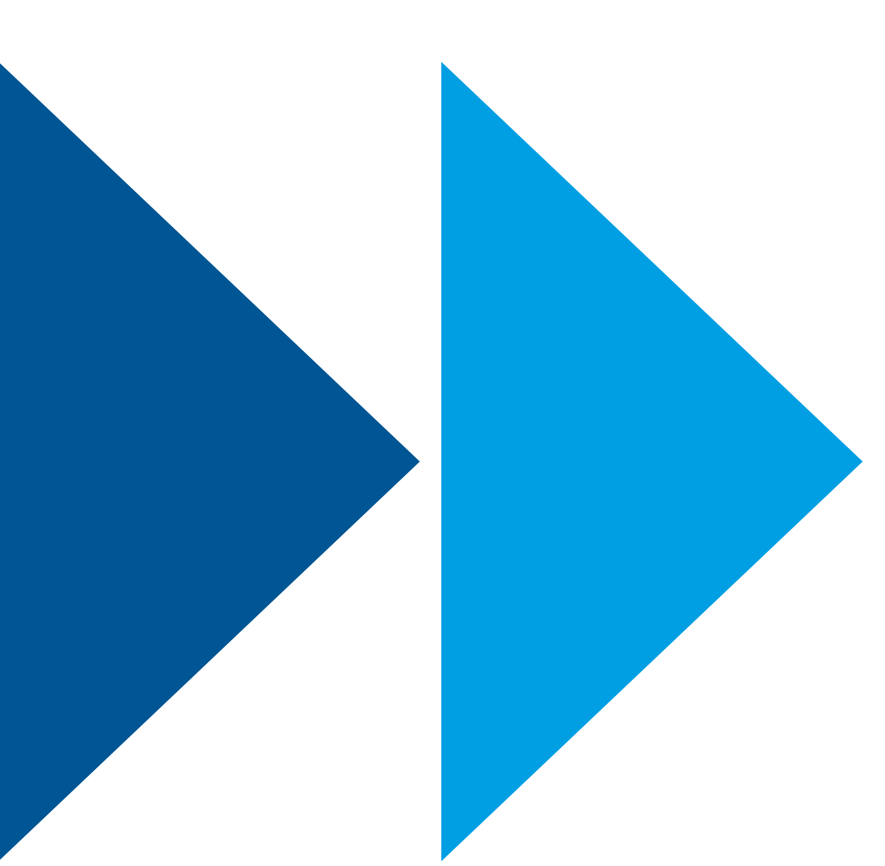
Lecture #03
Polymorphism

Today

Lecture #04
STL Containers

Lecture #05
Templates

Lecture #06
Exceptions



AGENDA

01 – What is Polymorphism ?

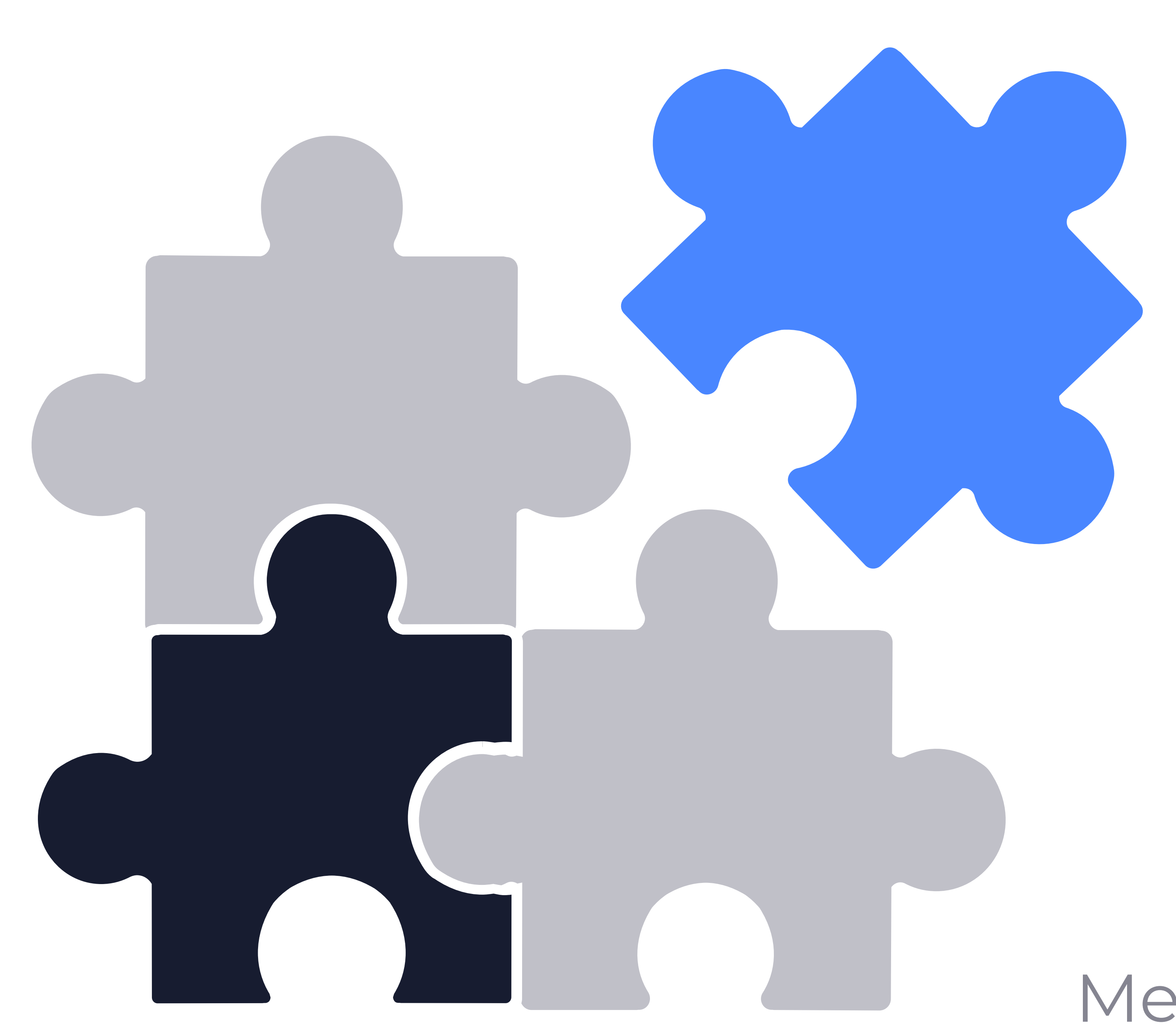
02 – Functions/methods overloading

03 – Operator overloading

04 – I/O overloading

05 – Abstract classes

The four pillars of OOP



Abstraction

Mechanism of hiding the implementation details from the user, only the functionality will be provided to the user.

Encapsulation

Mechanism, also known as Data hiding, that refers to the bundling of data/methods into a single coherent unit.

Inheritance

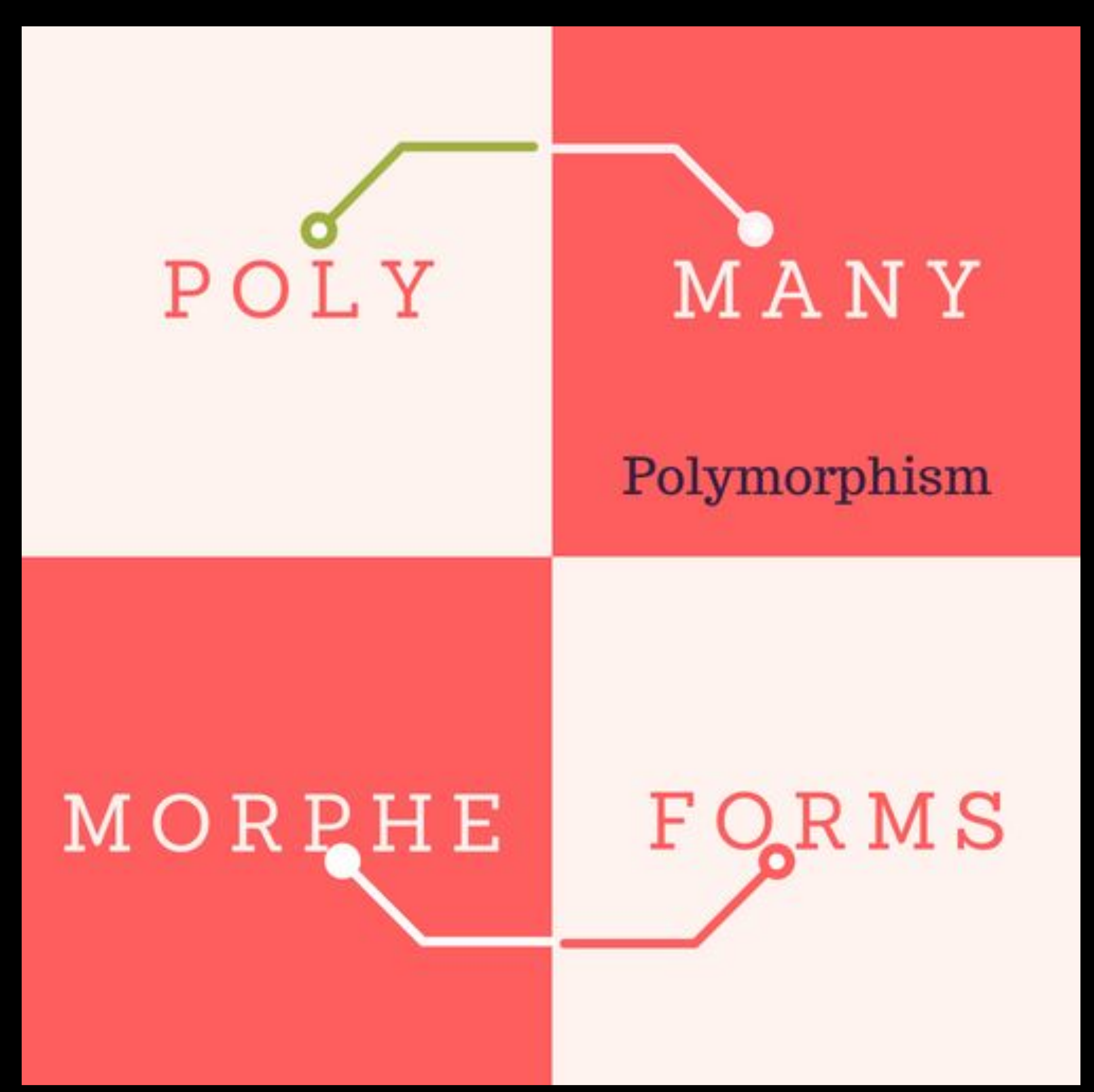
Mechanism of basing an object upon another object that allows sharing of properties and behaviors and implementation of new functionalities.

Polymorphism

Mechanism of assigning a different meaning or usage to something in different contexts.

Includes Overloading and Overriding.

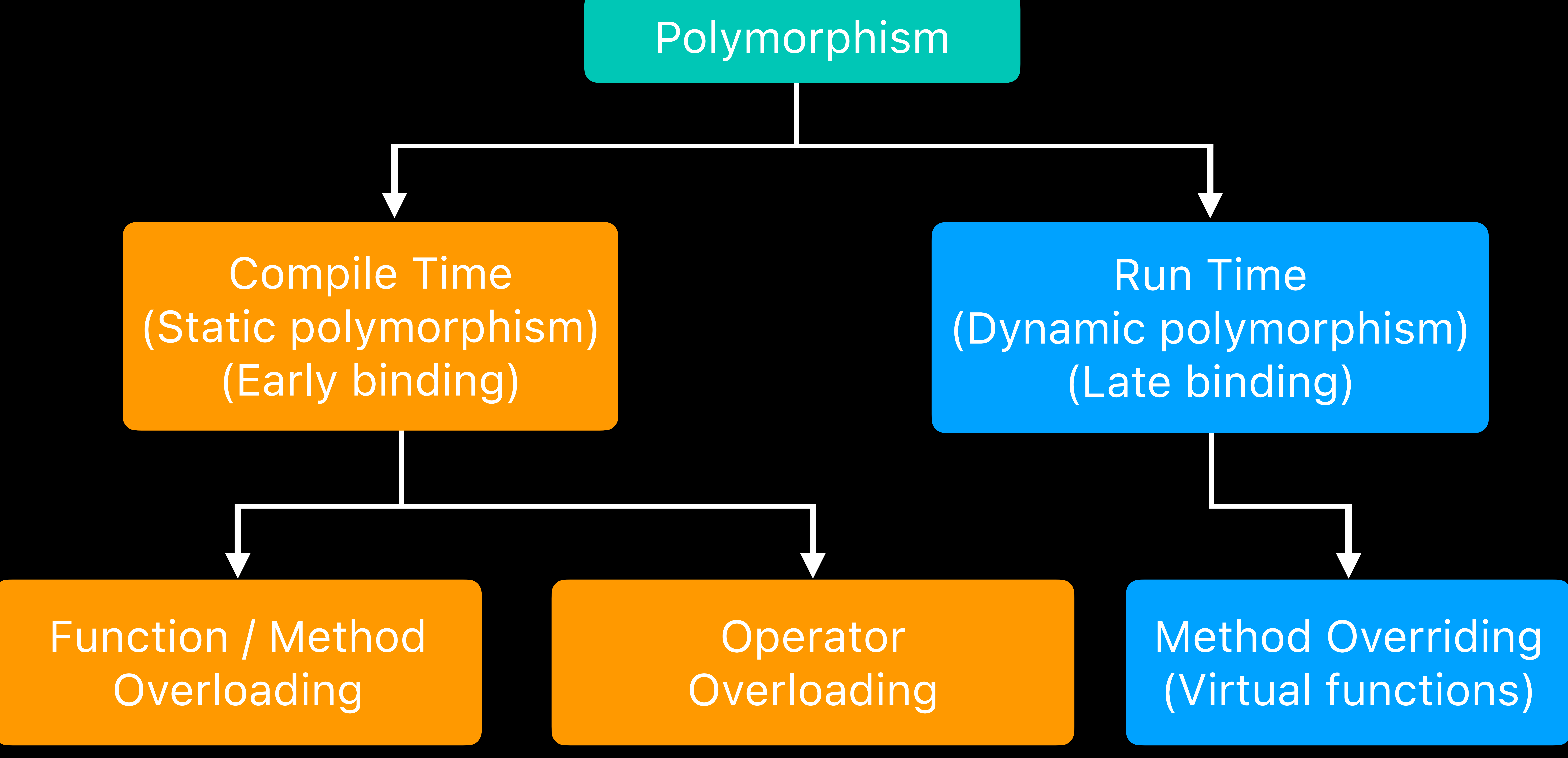
What is Polymorphism?



Polymorphism is a greek word that means having many forms

In C++, Polymorphism allows an object to behave differently in different scenarios

Types of Polymorphism in C++



AGENDA

01 – What is Polymorphism ?

02 – Functions/methods overloading

03 – Operator overloading

04 – I/O overloading

05 – Abstract classes

Functions/ Methods Overloading

Overloaded functions or methods are C++ functions with the same name, but with different and unique parameters!

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

Same Name
Different list of parameters

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

Works for functions or methods

```
int sameFunction(int a);
double sameFunction(int a);
```

Compilation error!
return type is NOT considered when overloading

Free functions Overloading

functions.cpp

```
namespace myfct {
  int absolute(int value) {
    if (value < 0)
      value = -value;
    return value;
  }
  float absolute(float value) {
    // Immediate if - Conditional operator
    return value < 0 ?  -value : value; // var = condition ? true : false;
  }
}
```

main.cpp

```
#include "functions.h"
int main(int argc, char const *argv[]) {
   int a = -5;  float b = 5.5;
   std::cout << "Abs(" << a << ") = " << myfct::absolute(a) << std::endl;
   std::cout << "Abs(" << b << ") = " << myfct::absolute(b) << std::endl;
   return 0;
}
```

-zsh — d0m ⌥⌘1

```
➜  basic-functions make
clang++ -Wall  -std=c++17 -MMD -c  functions.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app functions.o main.o
➜  basic-functions ./app
Abs(-5) = 5
Abs(5.5) = 5.5
➜  basic-functions
```

Method Overloading

Two public methods for updateDueDate declared in todo.h and defined in todo.cpp

```
class Todo {
   public:
      void updateDueDate(date::Date due_date);
      void updateDueDate(int days);
   private:
      date::Date _due_date;
   }; // End of Todo
```

```
void Todo::updateDueDate(date::Date due_date) {
   _due_date = due_date;
   }
void Todo::updateDueDate(int days) {
   for (int i=0; i<days; i++) {
      _due_date.next();
   }
}
```

Method Overloading

main.cpp

```
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(10,5);
    Category category = Category::Personal;
    int priority = LOW;
    todo::Todo todo1(title, category, priority, due_date);
    display(todo1);
    todo1.updatePriority(NORMAL));
    todo1.updateDueDate(date::Date(10,11));
    display(todo1);
    todo1.updatePriority(HIGH));
    todo1.updateDueDate(5);
    display(todo1);
    return 0;
}
```

-zsh — d0m ⌥⌘1

```
➜  methods-overloading make
clang++ -Wall  -std=c++17 -MMD -c  todo.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
➜  methods-overloading ./app
TODO: Buy Beer (Low priority - 5/10)
TODO: Buy Beer (Normal priority - 11/10)
TODO: Buy Beer (High priority - 16/10)
➜  methods-overloading
```

Questions

AGENDA

01 – What is Polymorphism ?

02 – Functions/methods overloading

03 – Operator overloading

04 – I/O overloading

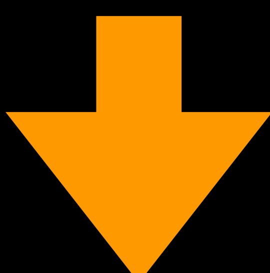
05 – Abstract classes

Operator overloading

C++ OPERATOR OVERLOADING allows operators to have user-defined meanings on classes

By overloading standard operators on a class, you can exploit the intuition of the users of that class

```
calculation = add(divide(a, b),multiply(a, b));
```



```
calculation = (a/b)+(a*b);
```

Operator overloading: why?

The basic purpose of operator overloading is used to provide facility to the programmer, to WRITE expressions in the most NATURAL form.

This lets users program in the language of the problem domain rather than in the language of the machine

2 + 3 = 5
Date(1972,5,26) + 3 = Date(1972,5,29)

2 < 3
Date(1972,5,26) < Date(1972,5,29)

Operator overloading: how?

Writing an Overload for the < operator

my_object < something

MyClass is on the left of operator < and the type of something (on the right) can be any type including MyClass (e.g. comparing 2 Date objects)

1 - Operator overloading = Method

```
bool myClass::operator<(OtherType something)
```

something < my_object

MyClass is on the right of operator < and you do not control the type of something (on the left of operator <).

2 - Operator overloading = Helper function

```
bool operator<(OtherType something, MyClass my_object)
```

3 - Operator overloading = Friend function

```
friend bool operator<(OtherType something, MyClass my_object)
```

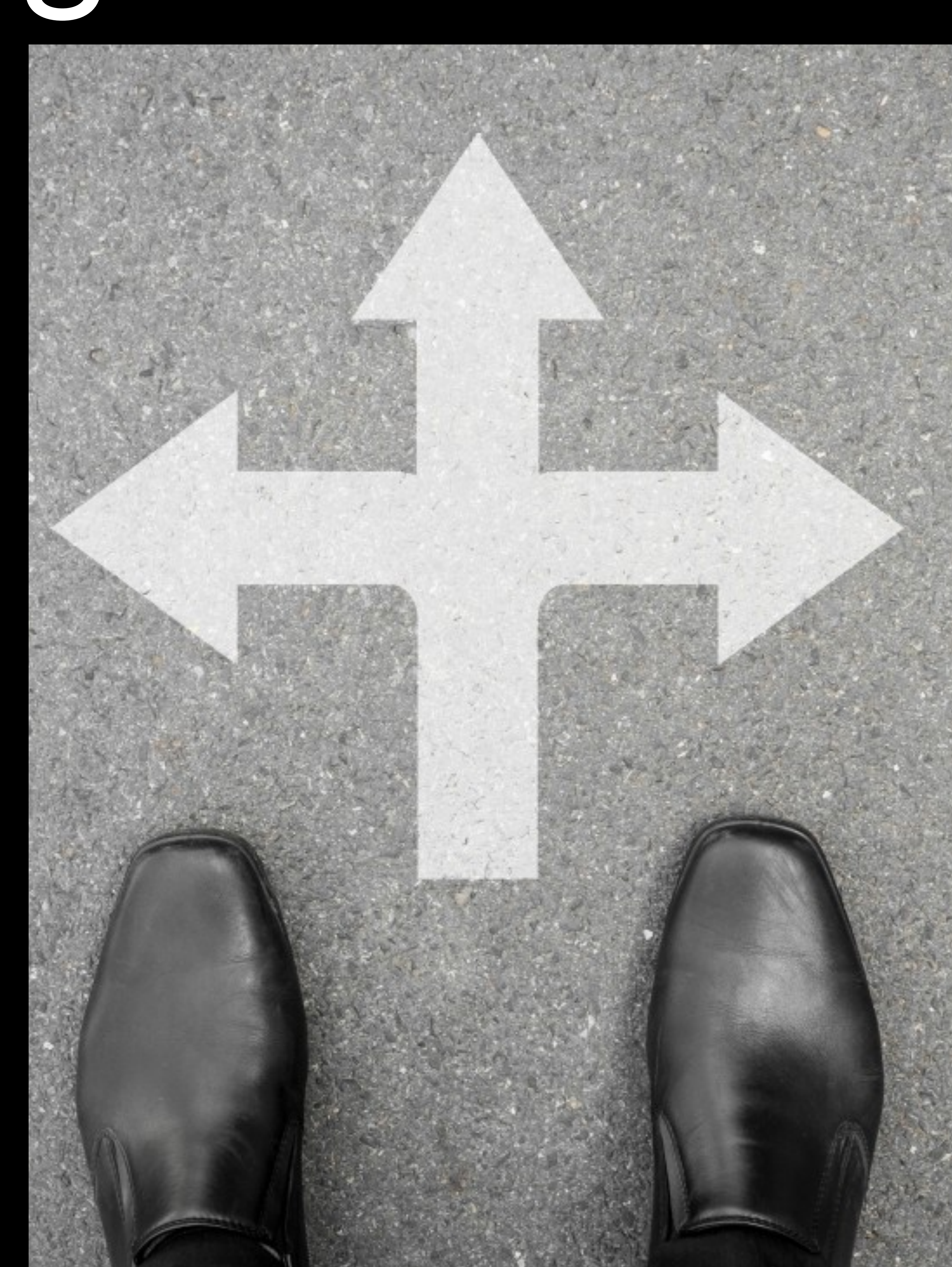
Operator overloading: how?

So, 3 different ways:

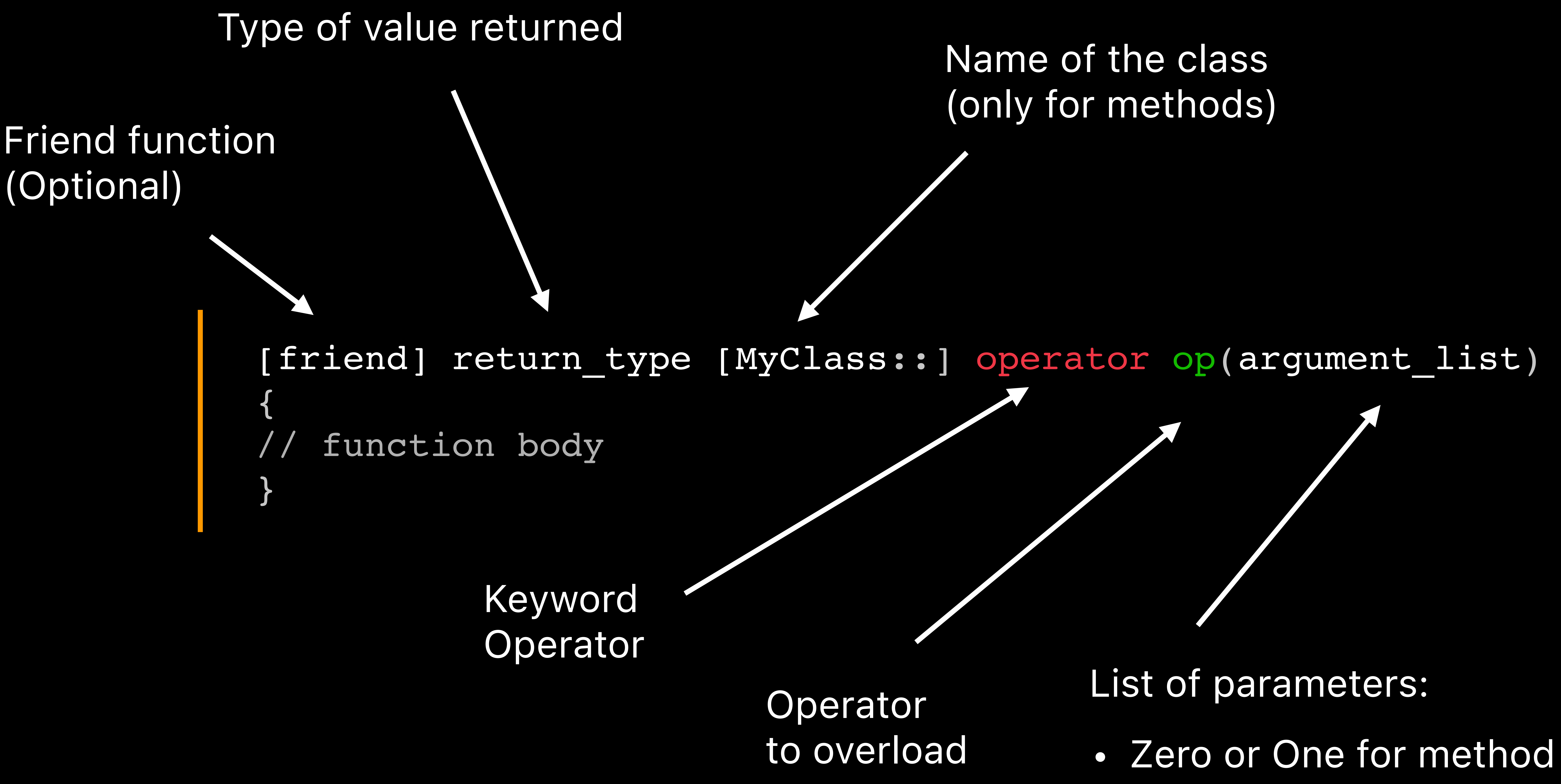
METHOD only if the left operand is an object of that class

HELPER function if the left operand of that particular class is an object of a different class or an object of the class

FRIEND function when we need to access the private members of a class without having getters



Operator overloading: syntax



Method, helper or friend?

When using method, expressions like s1+s2 are interpreted as s1.operator+(s2). SYMMETRIC operations such as s2+s1 with different types for s1 and s2 are not allowed (Ex: date + X days)

So, If you need symmetric operations, prefer implementing overloading operators with HELPER FUNCTIONS

Use friend function otherwise!

With methods

```
class Date {
  private:
    int _month;
    int _day;
  public:
    bool operator == (const Date& d) const;
    bool operator != (const Date& d) const; // d1 != d2
    bool operator < (const Date& d) const;  // d1 <  d2
    bool operator > (const Date& d) const;  // d1 >  d2
    bool operator <= (const Date& d) const; // d1 <= d2
    bool operator >= (const Date& d) const; // d1 <= d2
    Date operator + (int days) const; // d1 + integer
    Date operator - (int days) const; // d1 - integer
    Date operator ++ (); // ++date (prefix increment)
    Date operator -- (); // —date (prefix decrement)
    // The int gives information to the compiler that it is the postfix version
    Date operator ++ (int); // date++ (postfix increment)
    Date operator -- (int); // date-- (postfix decrement)
};
```

`const Date& d` is a reference

Code for all the operators is available on github

date.h

Operators == and !=

date.cpp

```
bool Date::operator == (const Date& d) const { // check for equality
    if ( (_day==d.day()) && (_month==d.month())) {
      return true;
    }
    return false;
}
bool Date::operator !=(const Date& d) const {
    return !(*this==d);
}
```

main.cpp

```
int main(int argc, char const *argv[]) {
   date::Date end_of_world(12,12);
   date::Date other_date(12,25);
   bool test = other_date == end_of_world;
   std::cout << "25/12 == End of World ? " << std::to_string(test) << '\n';
   test = other_date !=  end_of_world;
   std::cout << "25/12 != End of World ? " << std::to_string(test) << '\n';
   return 0;
}
```

-zsh — d0m

```
→  op-overloading-with-methods make
clang++ -Wall  -std=c++17 -MMD -c  todo.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
→  op-overloading-with-methods ./app Equal
25/12 == End of World ? 0
25/12 != End of World ? 1
→  op-overloading-with-methods
```

Operator +

date.cpp

```
Date Date::operator + (const int days) const {
      if (days <0) { //if days <0, we call Date - (-days)
         return *this - (-days);
      }
      Date tmp = *this; // the current date
      int new_day = tmp._day + days; // the new day is ok if new_day < end of month
      int new_month = tmp._month;
      int days_in_month = getDaysInMonth(tmp._month);
      while (new_day > days_in_month) { // end of the month
         new_day -= days_in_month; // the day in the next month
         new_month++;
         if (new_month > 12) { // end of the year
            new_month = 1;
        }
         tmp.updateMonth(new_month);
         days_in_month = getDaysInMonth(tmp._month);
      }
      return Date(new_month, new_day);
   }
```

Using operators +, -

main.cpp

```cpp
int main(int argc, char const *argv[]) {
    date::Date end_of_world(12,12);
    date::Date other_date(12,25);
    date::Date new_date = end_of_world + 70;
    std::cout << "12/12 + 70 days : " << toString(new_date) << '\n';
    new_date = end_of_world - 152;
    std::cout << "12/12 - 152 days : " << toString(new_date) << '\n';
    new_date = new_date + 50;
    std::cout << "12/12 - 152 + 50 days : " << toString(new_date) << '\n';
    return 0;
}
```

-zsh — d0m ⌥⌘1

```
➜  op-overloading-with-methods make
clang++ -Wall  -std=c++17 -MMD -c  todo.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
➜  op-overloading-with-methods ./app Arithmetic
12/12 + 70 days : 20/2
12/12 - 152 days : 13/7
12/12 - 152 + 50 days : 1/9
➜  op-overloading-with-methods
```

Operators ++ and —

date.cpp

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

```
d2=d1;
d1++;
return d2;
```

```
d2=d1;
d1—;
return d2;
```

```
++d1;
return d1;
```

```
--d1;
return d1;
```

Using Member Operators ++ and --

main.cpp

```
int main(int argc, char const *argv[]) {
  date::Date new_date = date::Date(02, 28);
  new_date++;
  new_date—;
  ++new_date;
  --new_date;

  new_date = date::Date(12, 31);
  date::Date tmp;
  tmp = ++new_date;
  tmp = --new_date;
  tmp = new_date++;
  tmp = new_date—;
  return 0;
}
```

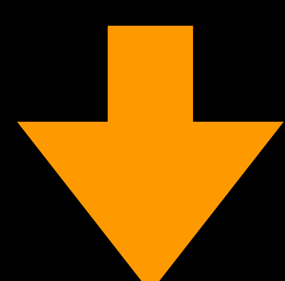
```
-zsh — d0m
➜  op-overloading-with-methods make
clang++ -Wall  -std=c++17 -MMD -c  todo.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
➜  op-overloading-with-methods ./app Inc
new_date=28/2
new_date++=1/3
new_date--=28/2
++new_date=1/3
--new_date=28/2

new_date = 31/12
tmp = ++new_date -> tmp=1/1 new_date=1/1
tmp = --new_date -> tmp=31/12 new_date=31/12
tmp = new_date++ -> tmp=31/12 new_date=1/1
tmp = new_date-- -> tmp=1/1 new_date=31/12
➜  op-overloading-with-methods
```

Helper functions

When using member functions, operations can't be SYMMETRICAL

date + integer is possible but integer + date can not be coded

```
Date Date::operator + (const int days) const; // date + integer
Date Date::operator - (const int days) const; // date - integer
```



Use HELPER functions defined outside the class

```
Date operator + (const Date& date, const int days); // date + integer
Date operator + (const int days, const Date& date); // integer + date
Date operator - (const Date& date, const int days); // date - integer
```

~~Date operator - (const int days, const Date& date);~~ // integer - date

Helper function +

date.cpp

```
Date operator + (const Date& date, int days) {
    if (days <0) { //if days <0, we call Date - (-days)
        return date - (-days);
    }
    Date tmp = date; // the current date
    int new_day = tmp.day() + days; // the new day is ok if new_day < end of month
    int new_month = tmp.month();
    int days_in_month = getDaysInMonth(tmp.month());
    while (new_day > days_in_month) { // end of the month
        new_day -= days_in_month; // the day in the next month
        new_month++;
        if (new_month > 12) { // end of the year
            new_month = 1;
        }
        tmp.updateMonth(new_month);
        days_in_month = getDaysInMonth(tmp.month());
    }
    return Date(new_month, new_day);
}
Date operator + (const int days, const Date& date) {
    return date + days;
}
```

All the operators can be rewritten using non member functions because getters are all available

Code available on github

Comparison of Operators == and !=

Helper functions

```
bool operator == (const Date& d1, const Date& d2)  { // check for equality
    if ((d1.day()==d2.day()) && (d1.month()==d2.month())) {
        return true;
    }
    return false;
}
bool operator !=(const Date& d1, const Date& d2)  {
    return !(d1==d2);
}
```

date.cpp

Methods

```
bool Date::operator == (const Date& d) const { // check for equality
    if ( (_day==d._day) && (_month==d._month)) {
      return true;
    }
    return false;
}
bool Date::operator !=(const Date& d) const {
    return !(*this==d);
}
```

Helper functions vs Methods

Comparison of Operators ++

Helper functions

```
Date Date::operator ++(Date& date, int) { // postfix increment
    Date tmp = date;
    date = tmp +1;
    return tmp;
}
Date Date::operator ++(Date& date) { // prefix increment
    date = date +1;
    return date;
}
```

date.cpp

Methods

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

Friend function?

To overload an operator with a non-member function, you must have access to the member variables through getters/setters.

```
bool operator == (const Date& d1, const Date& d2)  { // check for equality
  if ((d1.day()==d2.day()) && (d1.month()==d2.month())) {
      return true;
  }
  return false;
}
```

What happens if getters do not exist?

1. Make public the member variables? ....No, forbidden
2. Write getters/setters ? ....Possible but not always desirable
3. Use friend functions? ....Yes

Friend function

A friend function is declared in the class, giving it the right to ACCESS ALL PRIVATE members

But a friend function is NOT A MEMBER FUNCTION because it is implemented/defined out the class scope

```
class Date {
    private:
        int _month;
        int _day;
    public:  // friend function are declared in the class
        friend bool operator == (const Date& d1, const Date& d2); // d1 == d2
};
```

Friend function

A friend function is a helper function, i.e a non member function with a definition/implementation out the scope of the class but with a declaration in the class, allowing the access to the member variables

```
bool operator == (const Date& d1, const Date& d2)  { // check for equality
    if ((d1._day==d2._day) && (d1._month==d2._month)) {
        return true;
    }
    return false;
}
```

Using friend function is easy but care must be taken because friend functions compromise the ENCAPSULATION philosophy!

Comparison of Operators ==

Helper

```
bool operator == (const Date& d1, const Date& d2)  { // check for equality
    if ((d1.day()==d2.day()) && (d1.month()==d2.month())) {
        return true;
    }
    return false;
}
```

Member

```
bool Date::operator == (const Date& d) const { // check for equality
    if ( (_day==d._day) && (_month==d._month)) {
      return true;
    }
    return false;
}
```

date.cpp

Friend

```
bool operator == (const Date& d1, const Date& d2)  { // check for equality
    if ((d1._day==d2._day) && (d1._month==d2._month)) {
            return true;
        }
    return false;
    }
```

Helper vs Member vs Friend

Photo by Jon Flobrant on Unsplash

HOW

To choose between helper, member and friend for overloading operators ?

1. Design classes to be minimal to avoid complex code modification when a change to the representation is considered.
2. Always prefer a "helper operator" and use getters to access the representation of a class.
3. Make an overloaded operator a member only if it needs direct access to the representation of a class.
4. Use finally friend in other cases.

Questions ?

AGENDA

01 – What is Polymorphism ?

02 – Functions/methods overloading

03 – Operator overloading

04 – I/O overloading

05 – Abstract classes

Overloading I/O

C++ is able to input and output the built-in data types using the stream operators >> and <<.

>> and << operators also can be overloaded to perform input and output for user-defined types.

main.cpp

```
int main(int argc, char const *argv[]) {
    date::Date end_of_world(12,12);
    date::Date christmas(12,25);
    std::cout << toString(end_of_world) << std::endl;
    std::cout << christmas;
    return 0;
}
```

-zsh — d0m

```
➜  io-overloading-with-friend make
clang++ -Wall  -std=c++17 -MMD -c  todo.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
➜  io-overloading-with-friend ./app
Dec/12
Dec/25
➜  io-overloading-with-friend
```

Overloading I/O

Input/output Operators >> and << take a std::istream or std::ostream as the left hand argument (cout << my_string or cin >> my_string)

They can not be implemented as members functions and must be defined as HELPER or FRIEND functions

Friend

```
class Date {
      public:  // friend function is declared inside the class
      friend std::ostream& operator<<(std::ostream& os, const Date& date)
};
```

Helper

```
class Date {
};
// helper function is declared outside the class
std::ostream& operator<<(std::ostream& os, const Date& date);
```

Overloading I/O

Helper functions

```
std::ostream& operator<<(std::ostream& os, const Date& date)  {
    std::string month[12] = {"Jan", "Feb", "March", "April", "May", "June",
                             "July", "Aug", "Sept", "Oct", "Nov", "Dec"};
    std::string to_display = month[date.month()-1]
                             + "/" + std::to_string(date.day());
    os << to_display << std::endl;
    return os;
}
```

date.cpp

Friend

```
std::ostream& operator<<(std::ostream& os, const Date& date)  {
    std::string month[12] = {"Jan", "Feb", "March", "April", "May", "June",
                             "July", "Aug", "Sept", "Oct", "Nov", "Dec"};
    std::string to_display = month[date._month-1]
                             + "/" + std::to_string(date._day);
    os << to_display << std::endl;
    return os;
}
```

Questions

AGENDA

01 – What is Polymorphism ?

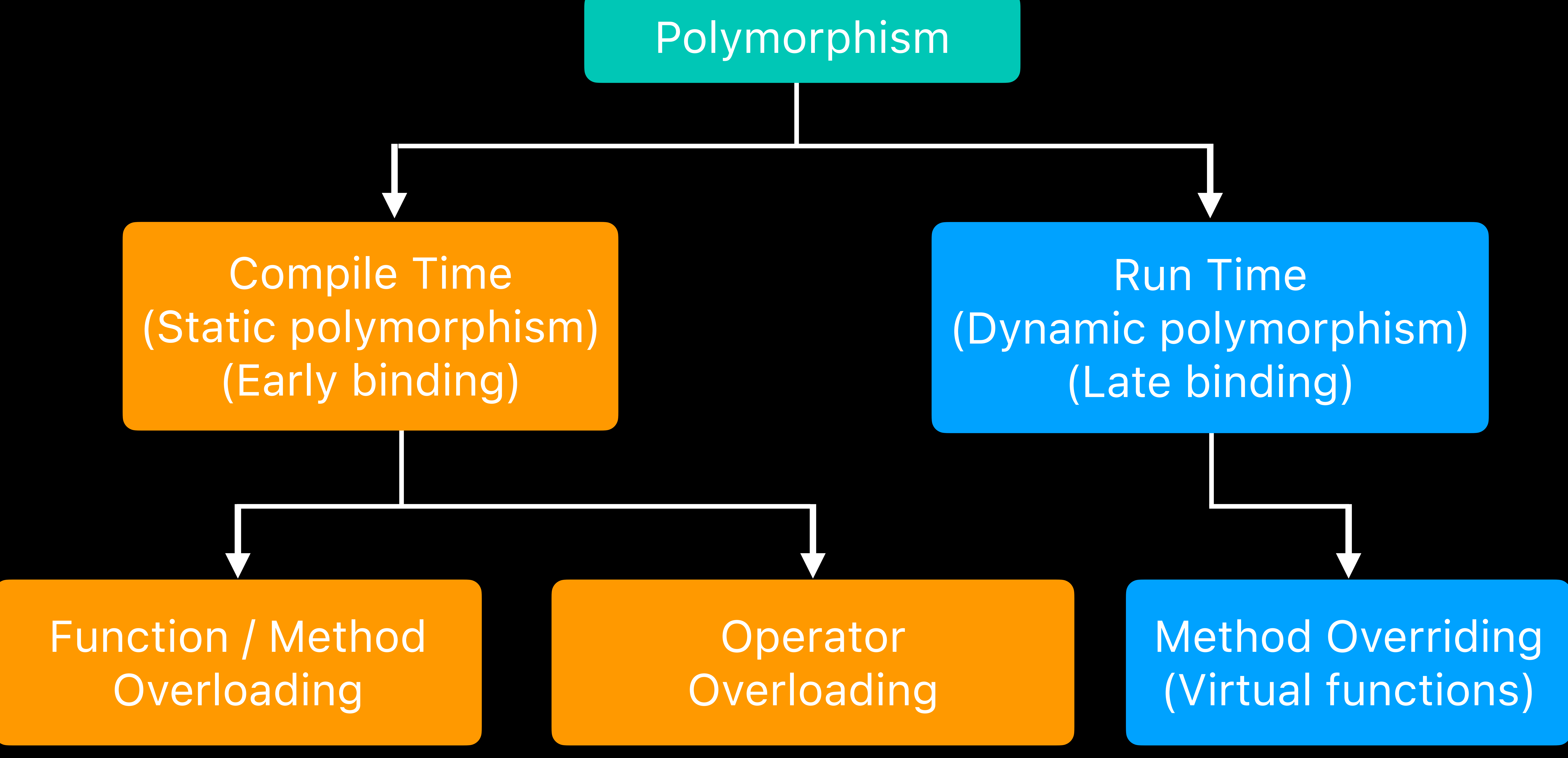
02 – Functions/methods overloading

03 – Operator overloading

04 – I/O overloading

05 – Abstract classes

Types of Polymorphism in C++



Static binding

Polymorphism

Compile Time
(Static polymorphism)
(Early binding)

Function / Method Overloading

Operator Overloading

STATIC BINDING means that the address of the code in a function invocation is checked at the earliest possible moment

Everything is known at compile time (no overhead)

Late binding

In Lecture #02, a RecurringTodo class was derived from a Todo class

However, we have not really OVERRIDDEN the base class methods

1. By redefining methods, we have just hidden the parent methods
2. C++ mechanisms that allow for POLYMORPHISM are not used

In some specific cases, the base class methods can be called, even on derived objects

Early binding - Demo

main.cpp

```
int main(int argc, char const *argv[]) {
   std::string title = "Play Piano";
   date::Date due_date(10,11);
   date::Date end_date(10,25);
   Category category = Category::Personal;
   int priority = NORMAL;
   int period = 3; // every 3 days
   todo::RecurringTodo r_todo1(title, category, priority,
                               due_date, period, end_date);
   display(r_todo1);
   todo::Todo *todo1 = &r_todo1;
   todo1->setCompleted(true);
   display(r_todo1);
}
```

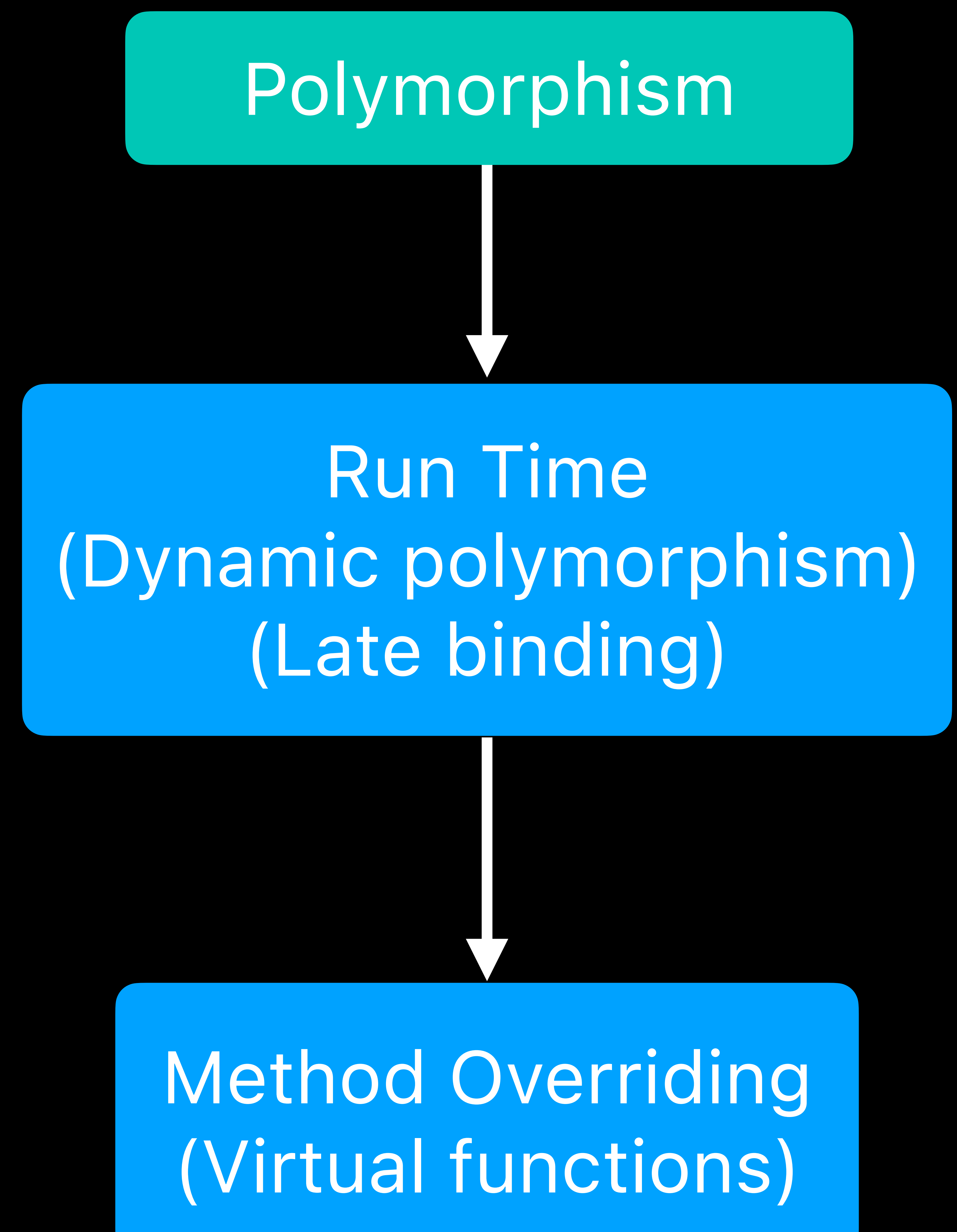
```
-zsh — d0m                                                    ⌥⌘3
→  early-binding make
clang++ -Wall  -std=c++17 -MMD -c   todo.cpp
clang++ -Wall  -std=c++17 -MMD -c   date.cpp
clang++ -Wall  -std=c++17 -MMD -c   main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
→  early-binding ./app
TODO: Play Piano (11/10) - every 3 days until 25/10
Todo setCompleted
DONE: Play Piano
→  early-binding
```

Dynamic or late binding

DYNAMIC or LATE BINDING means that the address of the function invocation is determined at the last possible moment (based on the dynamic type of the object at run time)

In C++, late binding is achieved by using the keyword virtual for functions

Internally, this is accomplished by creating a virtual table that contains one entry for each virtual function that can be called by objects of the class



Virtual function = Polymorphism

To use late binding (and polymorphism), we must explicitly tell the compiler that the function of the base class is overridden by the derived class

For this purpose, the function of the base class is then made ‘virtual’

```
class Todo {
   public:
      virtual void setCompleted(bool completed);
};
```

A virtual function is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class

Virtual function = Polymorphism

```
class RecurringTodo : public Todo {
    public:
        void setCompleted(bool completed) override;
}; // End of RecurringTodo
```

Since C++11, the override specifier(optional) in the derived class tells both the compiler (and also the reader) that the function is OVERRIDING a method from its base class

Using the override specifier is part of CLEAN CODING principles. It reveals the author's intentions, makes the code more readable and helps to identify bugs at build time. USE IT WITHOUT MODERATION!

Late binding - Demo

main.cpp

```
int main(int argc, char const *argv[]) {
   std::string title = "Play Piano";
   date::Date due_date(10,11);
   date::Date end_date(10,25);
   Category category = Category::Personal;
   int priority = NORMAL;
   int period = 3; // every 3 days
   todo::RecurringTodo r_todo1(title, category, priority,
                               due_date, period, end_date);
   display(r_todo1);
   todo::Todo *todo1 = &r_todo1;
   todo1->setCompleted(true);
   display(r_todo1);
}
```

-zsh — d0m

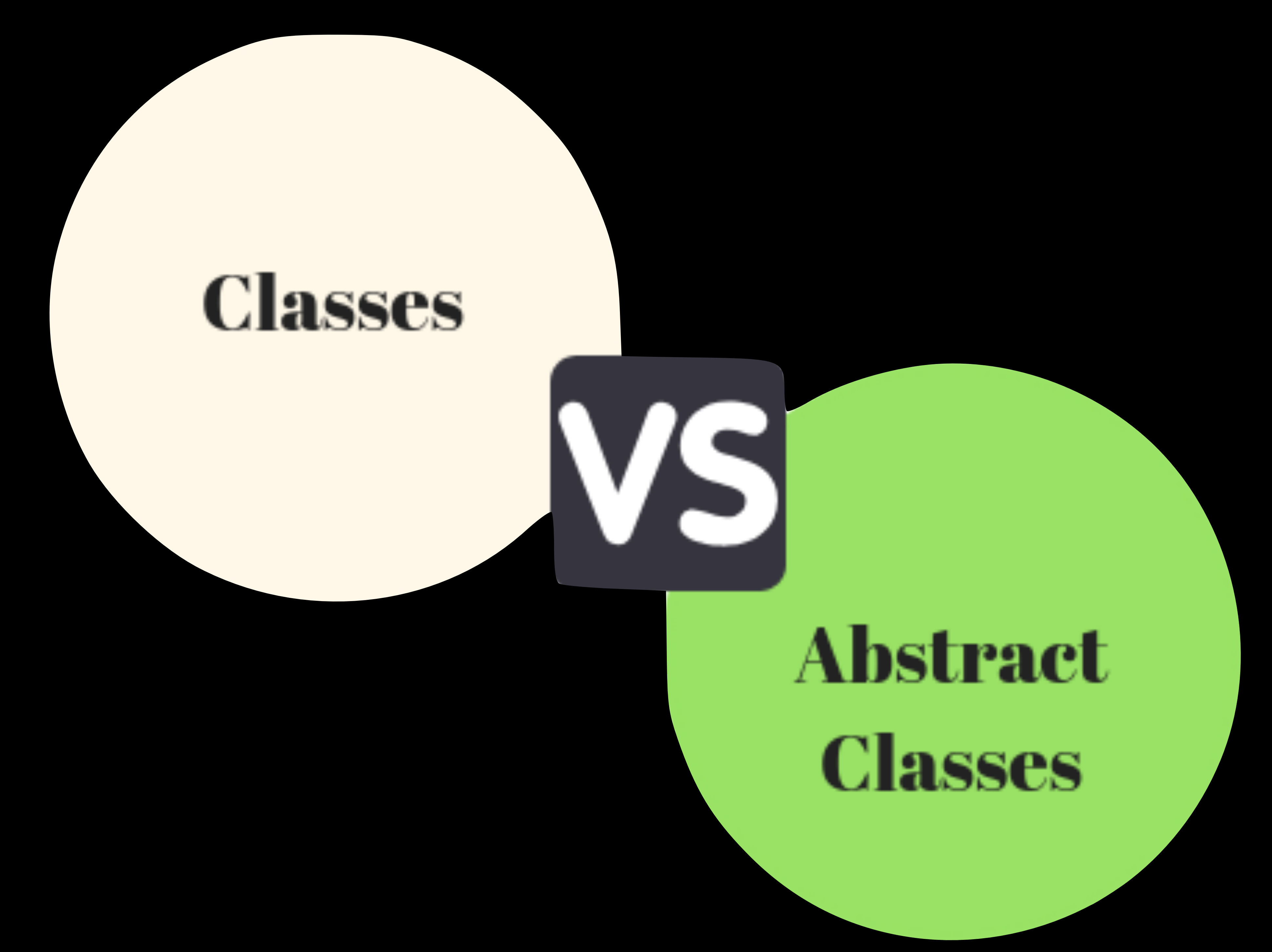
```
➜  late-binding make
clang++ -Wall  -std=c++17 -MMD -c  todo.cpp
clang++ -Wall  -std=c++17 -MMD -c  date.cpp
clang++ -Wall  -std=c++17 -MMD -c  main.cpp
clang++ -Wall  -std=c++17  -o app todo.o date.o main.o
➜  late-binding ./app
TODO: Play Piano (11/10) - every 3 days until 25/10
RecurringTodo setCompleted
TODO: Play Piano (14/10) - every 3 days until 25/10
➜  late-binding
```

Concept of Abstract class

Abstract classes are used to represent very GENERAL CONCEPTS (e.g. a Shape, an Animal), which can be used as base classes for more concrete classes (e.g. a Circle, a Dog)

Since Abstract classes serve as "blueprint" for derived classes, NO OBJECT of abstract classes can be created

Abstract classes are essential to provide ABSTRACTION to the code to make it reusable and extendable



How to create an Abstract class

An ABSTRACT class is a class that has at least ONE PURE VIRTUAL function, i.e. a function that has only a declaration (.h) and no definition (.cpp)

A pure virtual function simply acts as a placeholder that is meant to be overridden by derived classes

```
virtual void setCompleted(bool completed) = 0;
//=0 => PURE virtual
// No need to define it into .cpp file
// Only a declaration in .h file
```

An abstract class can't be instantiated because of the pure virtual functions

GenericTodo class

todo.h

```
class GenericTodo {
   public:
      GenericTodo(std::string title, Category category,
                  int priority, bool completed);
      virtual void setCompleted(bool completed) = 0;
  private:
      std::string _title;
      Category _category;
      int _priority;
  protected:
      bool _completed;
   };
```

Only COMMON variables are provided

Declared with only 1 constructor, 1 pure virtual functions, and the getters/setters for the member variables

Member variable _completed is protected

todo.cpp

```
GenericTodo::GenericTodo(std::string title, Category category,
                         int priority, bool completed) :
                         _title(title), _category(category),
                         _priority(priority), _completed(completed) {}
```

The new Todo class

The new Todo class inherits from GenericTodo, provides its constructor, adds a private due date variable with its getter/setter and the function that is virtual in the base class

```
class Todo : public GenericTodo {
   public:
      Todo(std::string title, Category category, int priority,
           date::Date due_date, bool completed=false);
      date::Date dueDate() const;
      void updateDueDate(date::Date due_date);
      void setCompleted(bool completed) override;
  private:
      date::Date _due_date;
   };
```

todo.h

The new RecurringTodo class

todo.h

```
class RecurringTodo: public GenericTodo {
   public:
      RecurringTodo(std::string title, Category category, int priority,
                    date::Date next_date, int period, date::Date end_date,
                    bool completed=false);
      date::Date nextDate() const;
      date::Date endDate() const;
      void updateNextDate(date::Date next_date);
      void updateEndDate(date::Date end_date);
      void setCompleted(bool completed) override;
      int period() const;
      void updatePeriod(int period) ;
  private:
      date::Date _next_date ;
      date::Date _end_date;
      int _period;
 };
```

Provides its CONSTRUCTOR

adds PRIVATE VARIABLES (next date, end date, period) with getter/setter

Provides also the IMPLEMENTATION of the virtual function

The setCompleted function

todo.cpp

```
void Todo::setCompleted(bool completed) {
      _completed = completed;
   }

void RecurringTodo::setCompleted(bool completed) {
      if (completed) {
         date::Date next_day = nextDate() + _period;
         updateNextDate(next_day);
         if (nextDate()>_end_date) {
            _completed = completed;
         }
      }
      else {
         _completed = false;
      }

   }
```

Acces to _completed is possible because _completed is protected

Using the two classes

Since GenericTodo is an abstract class, it's not possible to declare GenericTodo variables.

```
→  abstract-class make
clang++ -std=c++17 -MMD -c  main.cpp
main.cpp:20:4: error: allocating an object of abstract
class type 'todo::GenericTodo'
   todo::GenericTodo(title, category, priority);
   ^
./todo.h:37:20: note: unimplemented pure virtual method
 'setCompleted' in
     'GenericTodo'
     virtual void setCompleted(bool completed) = 0;
                  ^
1 error generated.
make: *** [main.o] Error 1
→  abstract-class
```

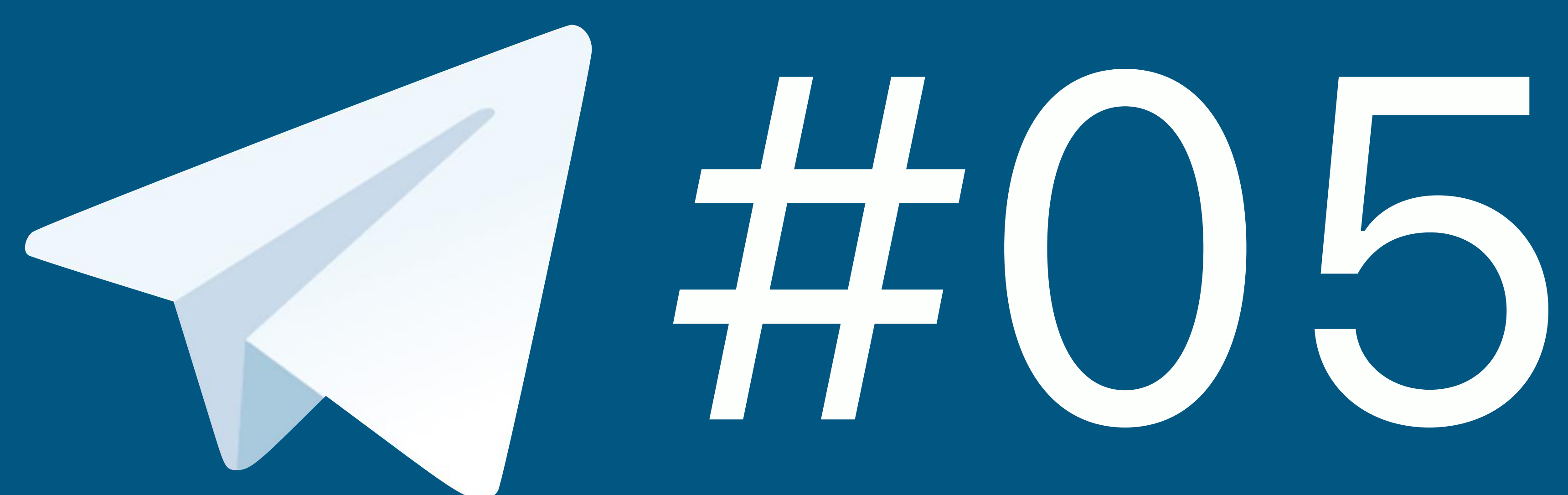
main.cpp

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(10,5);
    Category category = Category::Personal;
    int priority = NORMAL;

    todo::GenericTodo g_todo(title,
                             category,
                             priority);


    return 0;
}
```

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(10,5);
    Category category = Category::Personal;
    int priority = NORMAL;
    todo::Todo todo1(title, category,
                     priority, due_date);
    display(todo1);
    todo1.setCompleted(true);
    display(todo1);

    title = "Play Piano";
    date::Date next_date(10,11);
    date::Date end_date(11,4);
    int period = 3; // 3 days
    todo::RecurringTodo r_todo1(title, category, priority,
                                next_date, period, end_date);
    do {
      r_todo1.setCompleted(true);
      display(r_todo1);
    } while (!r_todo1.completed());
    return 0;
}
```

```
-zsh — d0m                                                    ⌥⌘3
→  abstract-class make
clang++ -std=c++17 -MMD -c  todo.cpp
clang++ -std=c++17 -MMD -c  date.cpp
clang++ -std=c++17 -MMD -c  main.cpp
clang++ -std=c++17  -o app todo.o date.o main.o
→  abstract-class ./app
TODO: Buy Beer (Normal priority - 5/10)
Todo setCompleted
DONE: Buy Beer (5/10)
TODO: Play Piano (14/10) - every 3 days until 4/11
TODO: Play Piano (17/10) - every 3 days until 4/11
TODO: Play Piano (20/10) - every 3 days until 4/11
TODO: Play Piano (23/10) - every 3 days until 4/11
TODO: Play Piano (26/10) - every 3 days until 4/11
TODO: Play Piano (29/10) - every 3 days until 4/11
TODO: Play Piano (1/11) - every 3 days until 4/11
TODO: Play Piano (4/11) - every 3 days until 4/11
DONE: Play Piano
→  abstract-class
```

main.cpp



#05

Take Home Message

POLYMORPHISM is one of the four core concepts of OOP

Overloaded functions / operators implemented as member, helper or friend functions are used to provide FACILITY to the programmer, to write expressions in the most natural form

Virtual functions and abstract class provides ABSTRACTION to the code making it reusable and extendable

Photo by Alexander Andrews on Unsplash

Contacts

Pr. Dominique Ginhac

dginhac@u-bourgogne.fr

Come visit us at

https://github.com/dginhac/esirem-itc313

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

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