Download slides on
https://github.com/dginhac/esirem-itc313

ITC313 - Lesson 04

Fundamentals of programming
Learning C++: from beginner to beyond

Dominique Ginhac

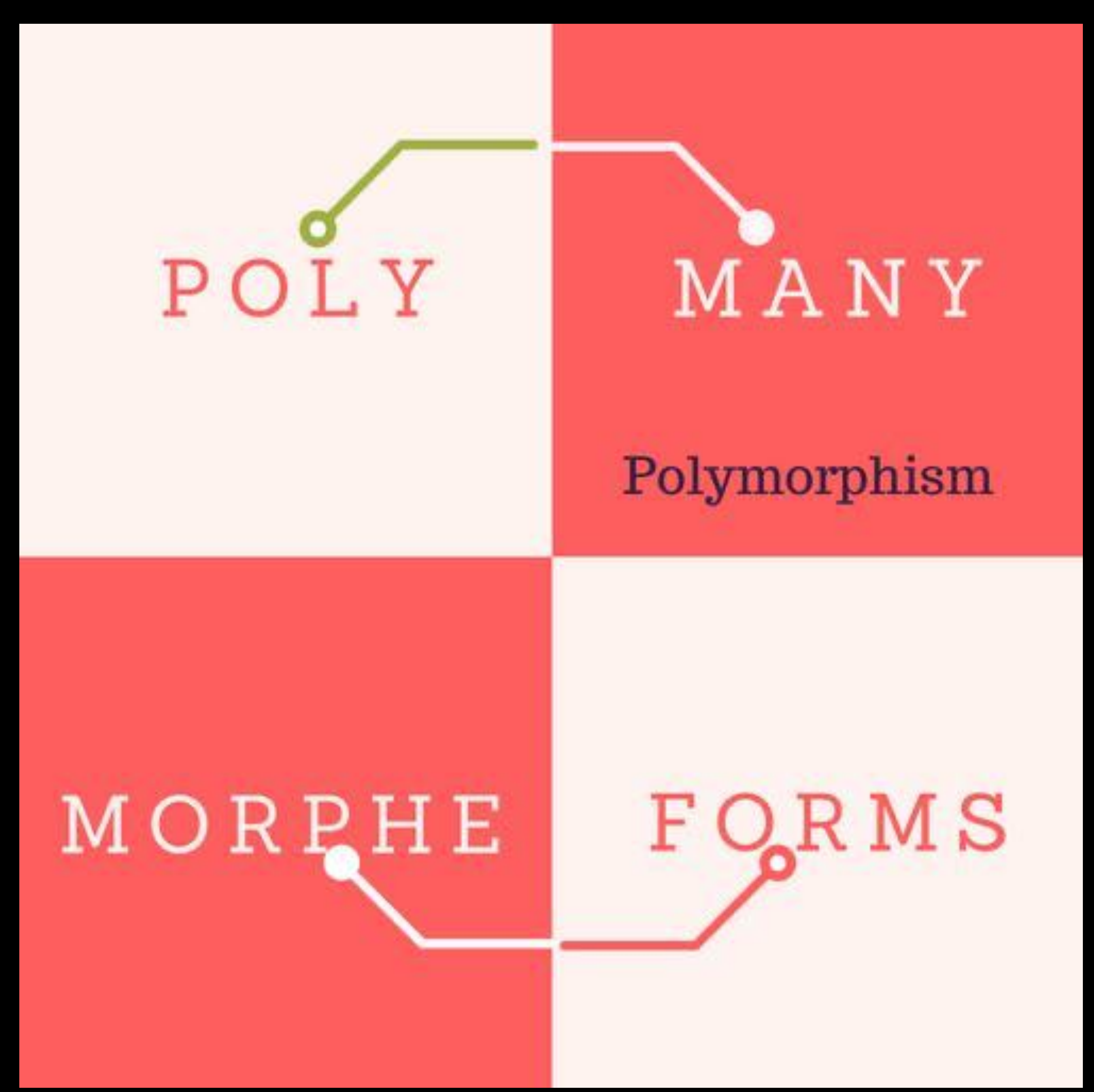
(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac

Lesson 04

Polymorphism

In this lesson, students will learn what Polymorphism is in C++. We will focus on method overriding and operator overloading and share examples of overriding / overloading common functions.

What is Polymorphism?

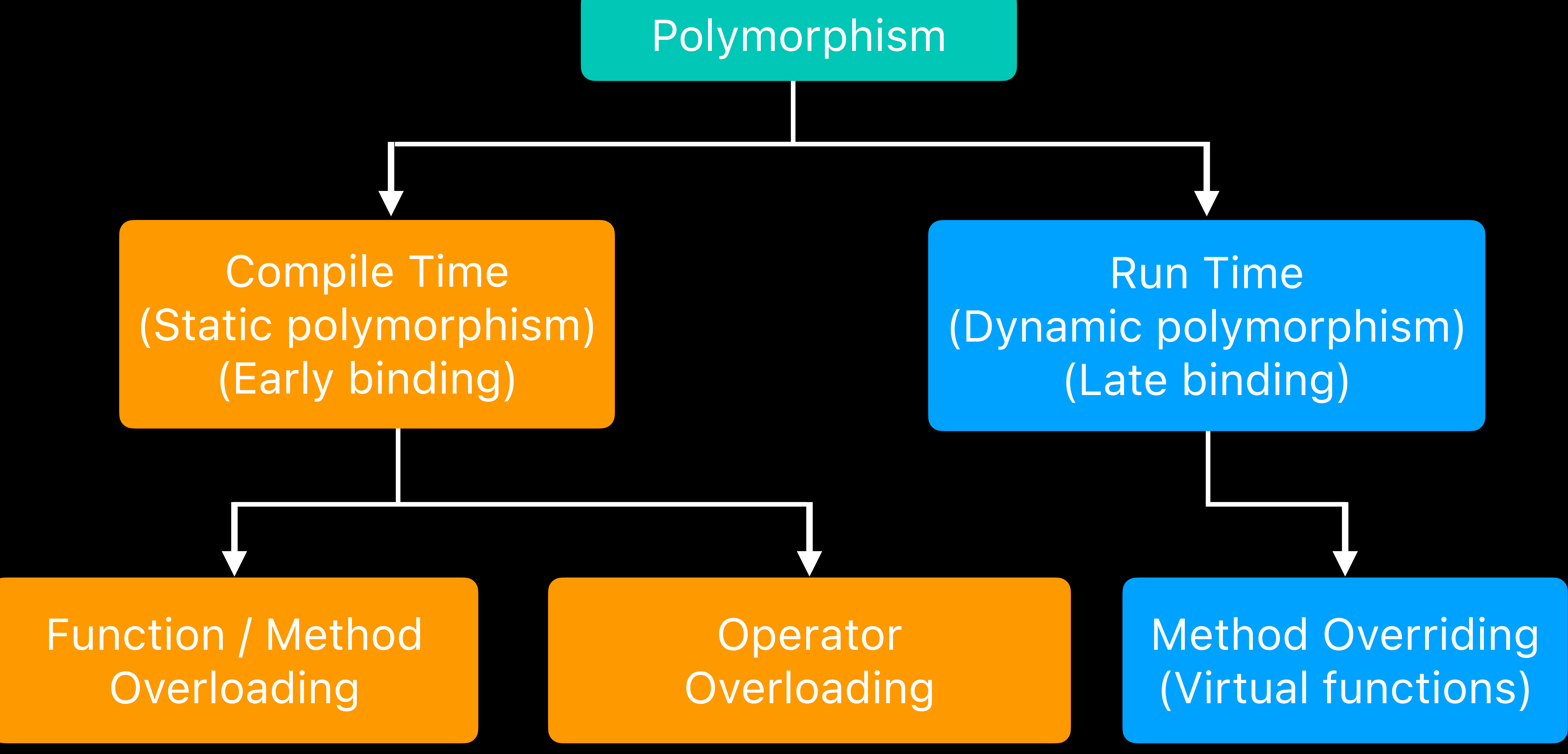


Polymorphism is a greek word that means having many forms

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

With abstraction, encapsulation and inheritance, it is the 4th core concepts of OOP

Types of Polymorphism in C++



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);

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

Same Name
Different parameters

Works for functions or methods

```
int BadFunction(int a);
int BadFunction(double a, int b);
```

Compilation error!
return type is NOT considered when overloading

Free functions Overloading

```cpp
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;
}

int main(int argc, char const *argv[]) {
    int a = -5;  float b = 5.5;
    std::cout << "Abs(" << a << ") = " << absolute(a) << std::endl;
    std::cout << "Abs(" << b << ") = " << absolute(b) << std::endl;
    a = 23;   b = -12.4;
    std::cout << "Abs(" << a << ") = " << absolute(a) << std::endl;
    std::cout << "Abs(" << b << ") = " << absolute(b) << std::endl;
    return 0;
}
```

lesson04 — -zsh — 78×7

```
➜  lesson04 clang++ -o overloading overloading.cpp
➜  lesson04 ./overloading
Abs(-5) = 5
Abs(5.5) = 5.5
Abs(23) = 23
Abs(-12.4) = 12.4
➜  lesson04
```

Method Overloading

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

```
class Todo : public GenericTodo {
   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.nextDay();
   }
}
```

Method Overloading

```
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,25);
    Category category = Category::Personal;
    int priority = HIGH;
    todo::Todo todo1(title, category, priority, due_date);
    todo1.display();
    todo1.updateDueDate(date::Date(2020,10,30));
    todo1.display();
    todo1.updateDueDate(10);
    todo1.display();
    return 0;
}
```

lesson04 — -zsh — 80×10

```
→  lesson04 make -f Makefile.todo
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  todo-main.cpp
clang++ -Wall  -std=c++14  -o todo-main date.o todo.o todo-main.o
→  lesson04 ./todo-main
TODO: Buy Beer (2020/Oct/25)
TODO: Buy Beer (2020/Oct/30)
TODO: Buy Beer (2020/Nov/9)
→  lesson04
```

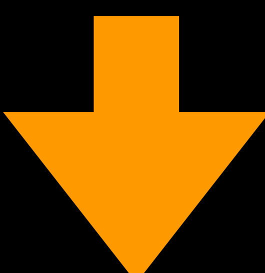
Operator overloading

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

OPERATOR OVERLOAD!

```
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

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

MyClass is on the left of operator <

MEMBER FUNCTION

OtherType can be any type, including MyClass (e.g. comparing 2 Date objects)

something < my_object

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

You do not control the type on the left of <

FREE FUNCTION

Use MyClass public functions for accessing variables

If not possible, exception with FRIEND FUNCTIONS

Operator overloading: how?

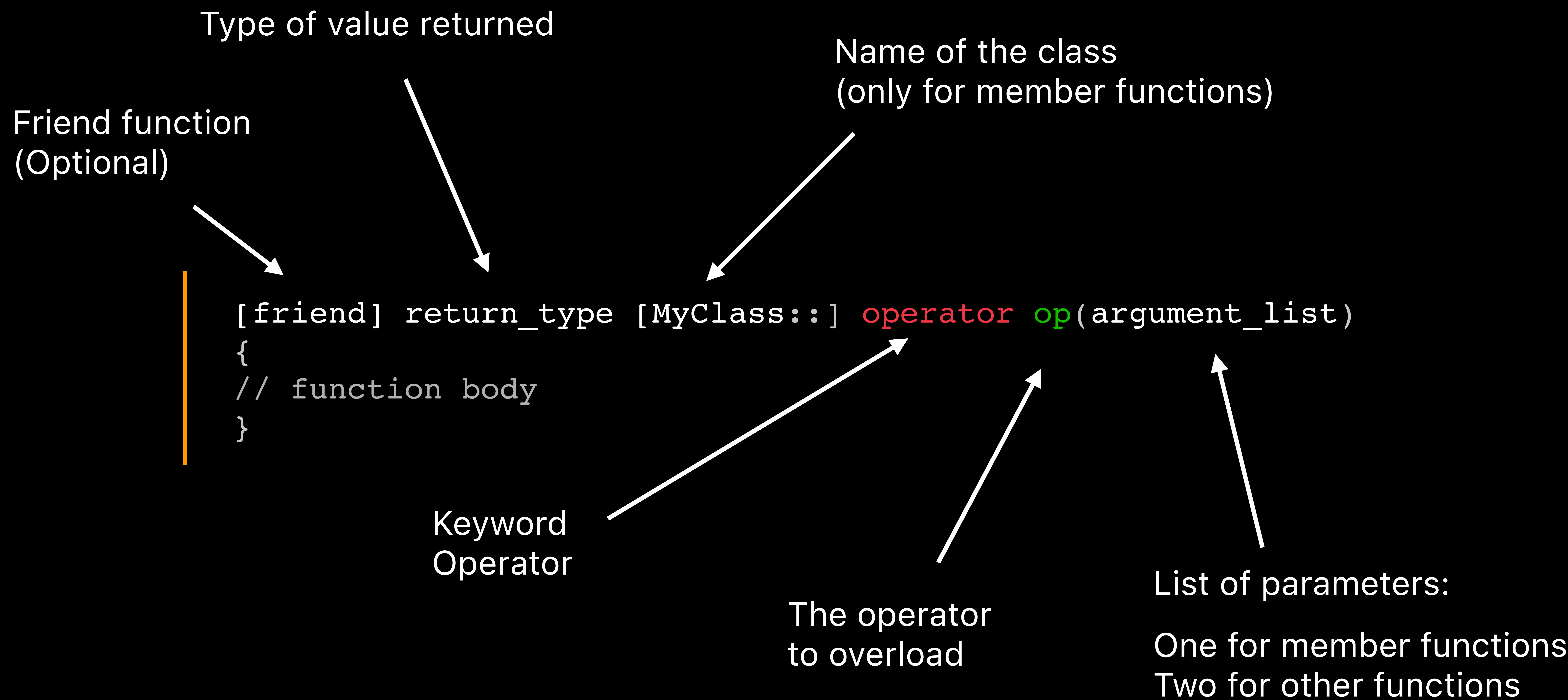
So, 3 different ways:

MEMBER function only if the left operand is an object of that class

NON-MEMBER function if the left operand of that particular class is an object of a different class

FRIEND functions when we need to access the private members of a class

Operator overloading: syntax



Member, free or friend?

When using member functions, 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

So, If you need symmetric operations, prefer implementing overloading operators with HELPER FUNCTIONS (i.e. free functions that obtain all the information from the public member functions)

Use friend function otherwise!

Member functions

```
class Date {
  private:
    int _year;
    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 + (const int days) const; // d1 + integer
    Date operator - (const 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

Upcoming lesson

Code for all the operators is available on github

Operators d1==d2? and d1!=d2

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

```cpp
int main(int argc, char const *argv[]) {
    date::Date end_of_world(2012,12,12);
    date::Date other_date(2020,12,25);
    bool test = other_date == end_of_world;
    std::cout << "25/12/2020 == End of World ? " << std::to_string(test) << '\n';
    test = other_date !=  end_of_world;
    std::cout << "25/12/2020 != End of World ? " << std::to_string(test) << '\n';
    return 0;
}
```

lesson04 — -zsh — 78×9

```
➜  lesson04 make -f Makefile.overload-op-members
clang++ -Wall  -std=c++14 -MMD -c  overload-op-members.cpp
clang++ -Wall  -std=c++14 -MMD -c  overload-op-members-main.
clang++ -Wall  -std=c++14  -o overload-op-members-main overl
verload-op-members-main.o
➜  lesson04 ./overload-op-members-main
25/12/2020 == End of World ? 0
25/12/2020 != End of World ? 1
➜  lesson04
```

Operator d2 = d1 + int

```
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 days_in_month = tmp.daysInMonth();
      int new_day = tmp.day() + days; // the new day is ok if new_day < end of month
      int new_month = tmp.month();
      int new_year = tmp.year();
      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;
            new_year++;
         }
         tmp.setMonth(new_month);
         days_in_month = tmp.daysInMonth();
      }
      return Date(new_year, new_month, new_day);
   }
```

Using operators +, -

```
int main(int argc, char const *argv[]) {
   date::Date end_of_world(2012,12,12);
   date::Date other_date(2020,12,25);
   date::Date new_date = end_of_world + 70;
   std::cout << "12/12/2012 + 70 days : " << new_date.toString() << '\n';
   new_date = end_of_world - 152;
   std::cout << "12/12/2012 - 152 days : " << new_date.toString() << '\n';
   return 0;
}
```

lesson04 — -zsh — 78×9

```
➜  lesson04 make -f Makefile.overload-op-members
clang++ -Wall  -std=c++14 -MMD -c  overload-op-members.cpp
clang++ -Wall  -std=c++14 -MMD -c  overload-op-members-main.cpp
clang++ -Wall  -std=c++14  -o overload-op-members-main overload-op-members.o o
verload-op-members-main.o
➜  lesson04 ./overload-op-members-main
12/12/2012 + 70 days : 2013/Feb/20
12/12/2012 - 152 days : 2012/July/13
➜  lesson04
```

Operators ++ and —

```
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;
}
```

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

d1--;
d2 = d1--;

++d1;
d2 = ++d1;

--d1;
d2 = --d1;
```

Using Member Operators ++ and --

```
int main(int argc, char const *argv[]) {
  date::Date new_date = date::Date(2020, 12, 31) ;
  date::Date tmp;
  tmp = ++new_date;
  tmp = --new_date;
  tmp = new_date++;
  tmp = new_date–;
  new_date++;
  ++new_date;
  new_date–;
  --new_date;
  return 0;
}
```

lesson04 — -zsh — 78×15

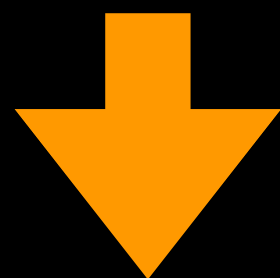
```
➜  lesson04 make -f Makefile.overload-op-members
clang++ -Wall  -std=c++14 -MMD -c  overload-op-members-main.cpp
clang++ -Wall  -std=c++14  -o overload-op-members-main overload-op-members.o o
verload-op-members-main.o
➜  lesson04 ./overload-op-members-main
2020/Dec/31
tmp = ++new_date -> tmp=2021/Jan/1 new=2021/Jan/1
tmp = --new_date -> tmp=2020/Dec/31 new=2020/Dec/31
tmp = new_date++ -> tmp=2020/Dec/31 new=2021/Jan/1
tmp = new_date-- -> tmp=2021/Jan/1 new=2020/Dec/31
new=2021/Jan/1
new=2021/Jan/2
new=2021/Jan/1
new=2021/Jan/2
➜  lesson04
```

Free functions

When using member functions, operations can’t be SYMMETRICAL

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

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



Use FREE functions defined outside the class

```
Date operator + (const Date& date, const int days); // date + integer
Date operator - (const int days, const Date& date); // d1 + integer
```

Free function +

```
Date operator + (const Date& date, const int days) {
    if (days <0) { //if days <0, we call Date - (-days)
        return date - (-days);
    }
    Date tmp = date; // the current date
    int days_in_month = tmp.daysInMonth();
    int new_day = tmp.day() + days; // the new day is ok if new_day < end of month
    int new_month = tmp.month();
    int new_year = tmp.year();
    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;
            new_year++;
        }
        tmp.setMonth(new_month);
        days_in_month = tmp.daysInMonth();
    }
    return Date(new_year, 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

Code available on github

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())&&(d1.year()==d2.year())) {
        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 out the class scope

```
class Date {
   private:
      int _year;
      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 free function, i.e a non member function with a declaration out the scope of the class

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

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

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.

```
int main(int argc, char const *argv[]) {
    date::Date end_of_world(2012,12,12);
    date::Date other_date(2020,12,25);
    std::cout << end_of_world.toString() << std::endl;
    std::cout << end_of_world;
    return 0;
}
```

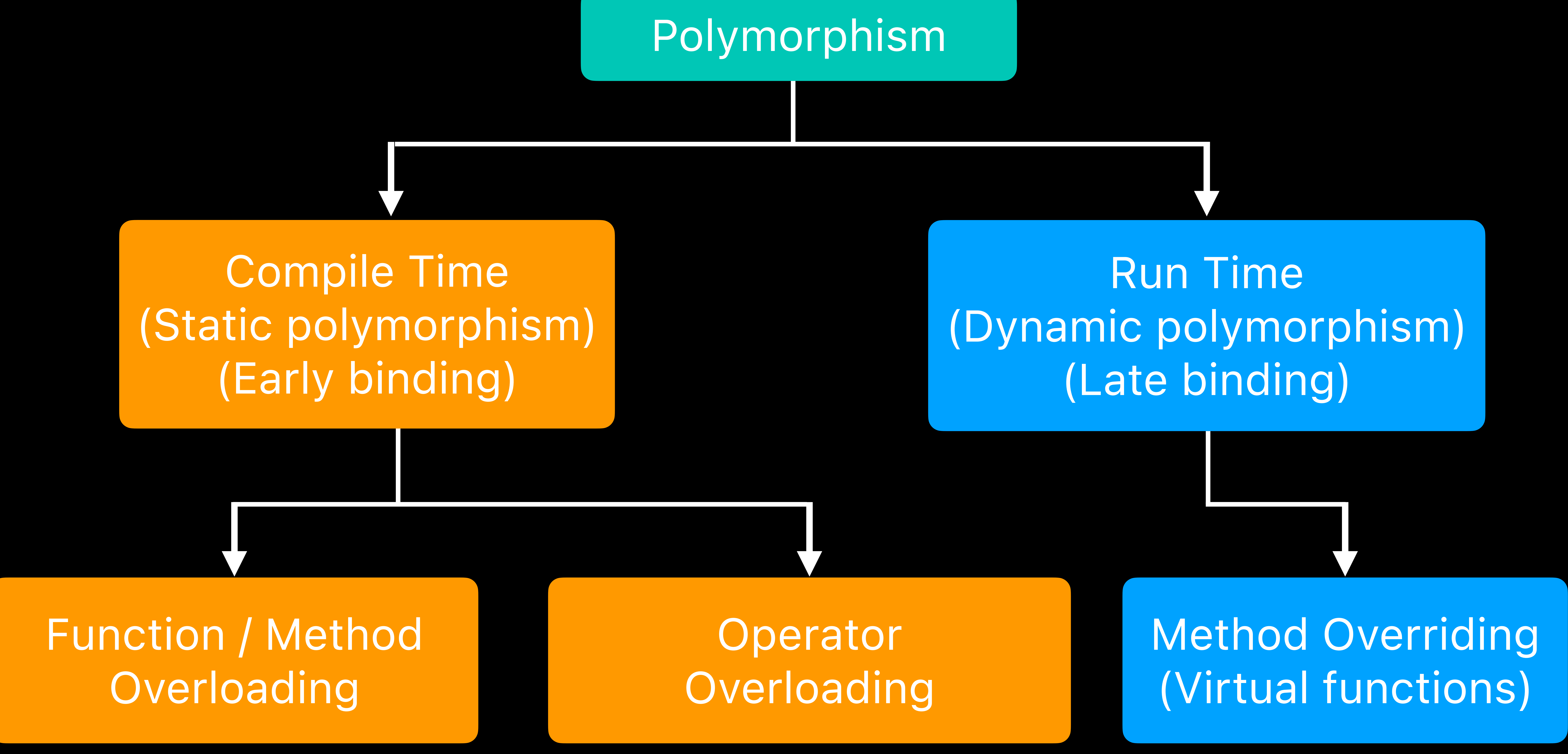
Overloading I/O

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 FREE or FRIEND functions (often as 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;
    to_display = std::to_string(date.year()) + "/"
                 + month[date.month()-1]
                 + "/" + std::to_string(date.day());
    os << to_display << std::endl;
    return os;
}
```

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 Lesson 03, 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

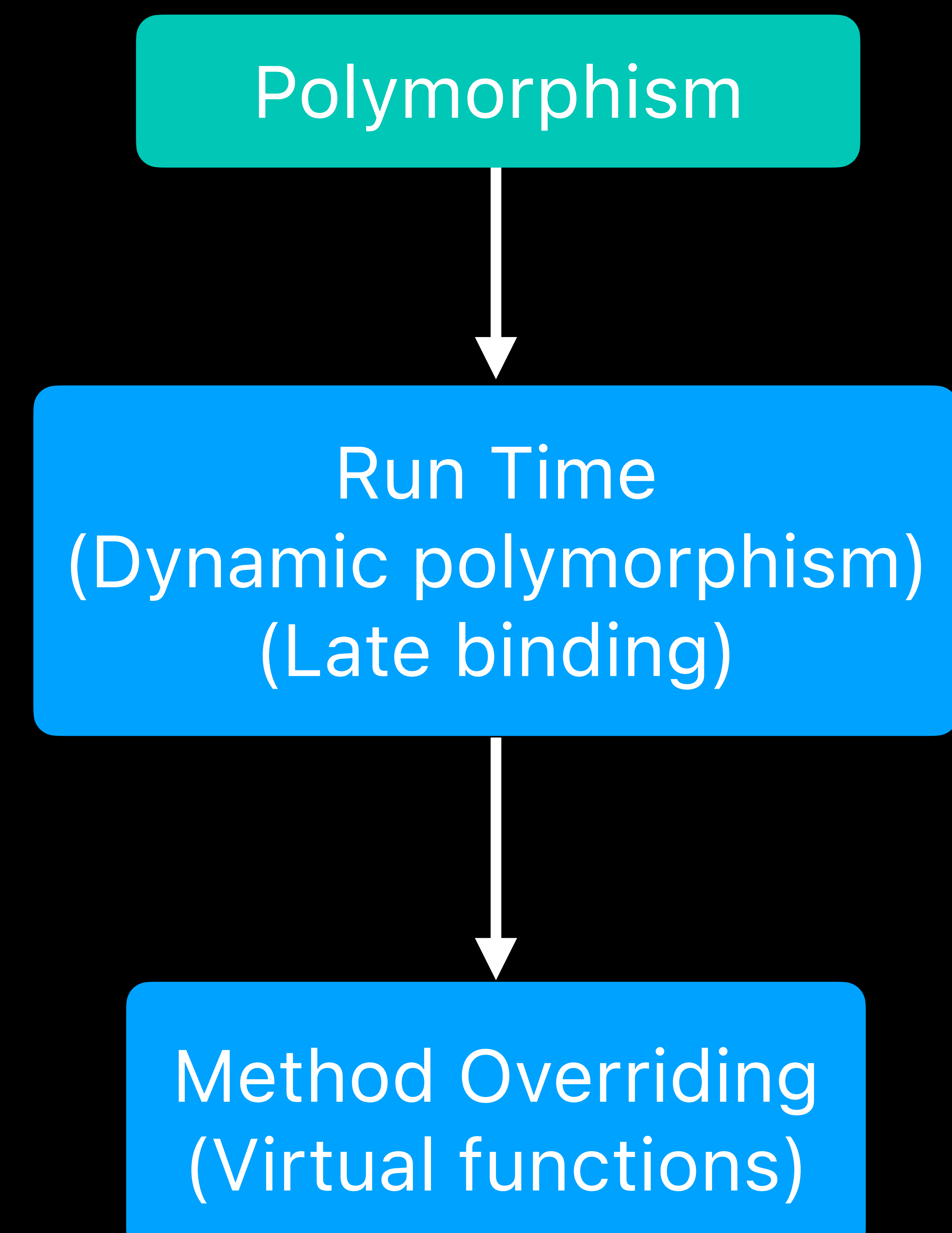
In some specific cases, the base class methods can be called, even on derived objects!

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 functions of the base class is then made ‘virtual’

```
class GenericTodo {
  public:
    virtual void setCompleted(bool completed);
    virtual void display() const;
};
```

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 Todo : public GenericTodo {
    public:
        void setCompleted(bool completed) override;
        void display() const override;
}; // End of Todo
```

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

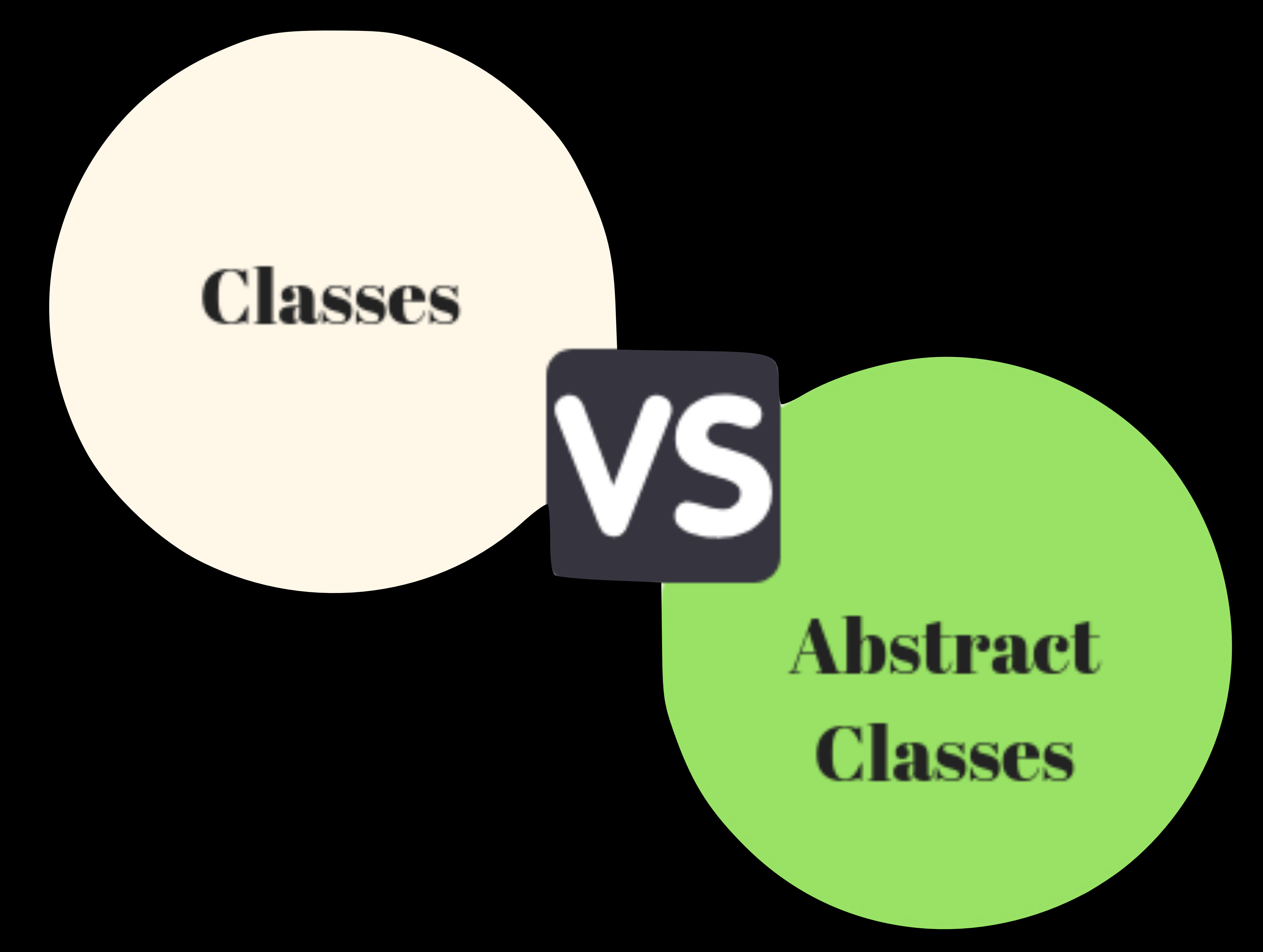
Using the override specifier is part of CLEAN CODING principles. It reveals the author's intentions, makes the code more readable and helps to identify bugs at build time. USE IT WITHOUT MODERATION!

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
virtual void display() const = 0; // No need to define it into .cpp file
```

An abstract class can’t be instantiated because of the virtual function

GenericTodo class

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

GenericTodo uses protected variables for reasons of SIMPLICITY

Only COMMON variables are provided

Declared with only 1 constructor and 2 pure virtual functions

```
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 two functions that are 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);
      void display() const;
   private:
      date::Date _due_date;
   };
```

The new RecurringTodo class

```
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);
        int period() const;
        void updatePeriod(int period);
        void display() const;
    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 two virtual functions

The setCompleted function

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

void RecurringTodo::setCompleted(bool completed) {
      if (completed) {
         for (int i=0; i<_period; i++) {
            _next_date.nextDay();
         }
         if (_next_date>_end_date) {
            _completed = completed;
         }
      }
      else {
         _completed = false;
      }

   }
```

With protected variables in the base class, writing functions in the derived class is easier.

But be careful not to change the name of the variables in the base class!

Using the two classes

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

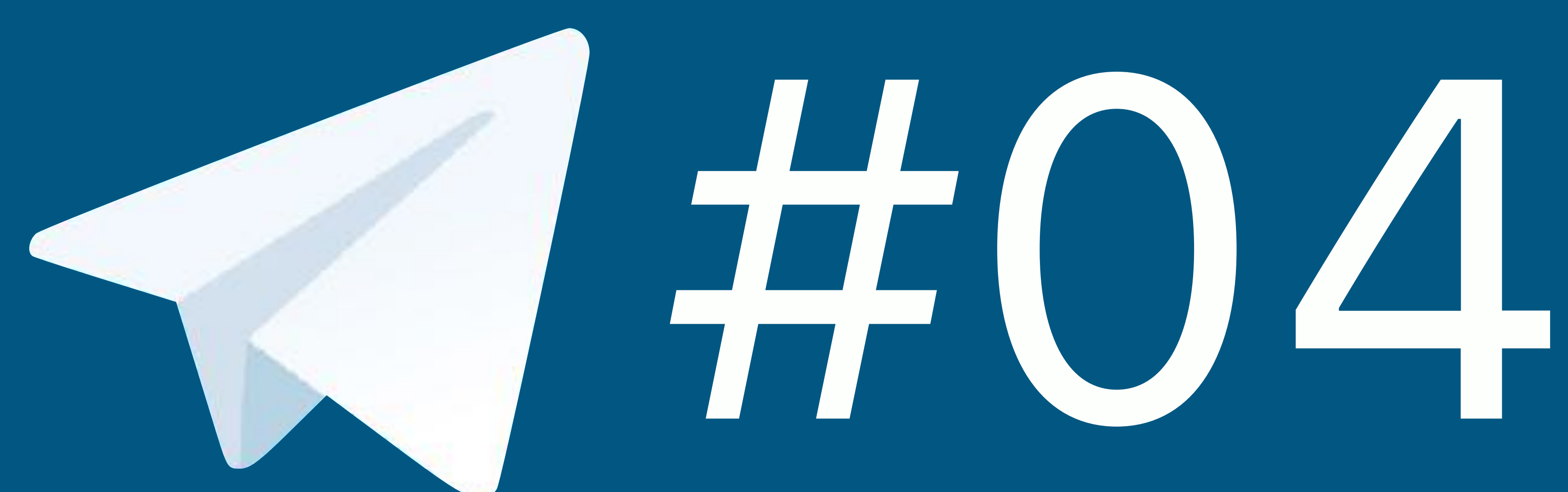
```
lesson03 — -zsh — 70×19
→ lesson03 make -f Makefile.abstract
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo-main.cpp
abstract-todo-main.cpp:25:22: error: variable type 'todo::GenericTodo'
      is an abstract class
   todo::GenericTodo g_todo(title, category, priority, due_date);
                     ^
./abstract-todo.h:25:20: note: unimplemented pure virtual method
      'setCompleted' in 'GenericTodo'
      virtual void setCompleted(bool completed) = 0;
                   ^
./abstract-todo.h:26:20: note: unimplemented pure virtual method
      'display' in 'GenericTodo'
      virtual void display() const = 0;
                   ^
1 error generated.
make: *** [abstract-todo-main.o] Error 1
→ lesson03
```

```cpp
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,25);
    Category category = Category::Personal;
    int priority = HIGH;

    todo::GenericTodo g_todo(title,
                             category,
                             priority,
                             due_date);
    g_todo.display();

    return 0;
}
```

```
int main(int argc, char const *argv[]) {
    std::string title = "Buy Beer";
    date::Date due_date(2020,10,25);
    Category category = Category::Personal;
    int priority = HIGH;
    todo::Todo todo1(title, category,
                     priority, due_date);
    todo1.display();
    todo1.setCompleted(true);
    todo1.display();

    title = "Play Piano";
    date::Date next_date(2020,10,20);
    date::Date end_date(2020,11,4);
    int period = 2; // 2 days
    todo::RecurringTodo r_todo1(title, category, priority,
                                next_date, period, end_date);
    do {
        r_todo1.setCompleted(true);
        r_todo1.display();
    } while (!r_todo1.completed());
    return 0;
}
```

lesson03 — -zsh — 62×18

```
→  lesson03 make -f Makefile.abstract
clang++ -Wall  -std=c++14 -MMD -c  date.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo.cpp
clang++ -Wall  -std=c++14 -MMD -c  abstract-todo-main.cpp
clang++ -Wall  -std=c++14  -o abstract-todo-main date.o abstra
ct-todo.o abstract-todo-main.o
→  lesson03 ./abstract-todo-main
TODO: Buy Beer (2020/Oct/25)
DONE: Buy Beer (2020/Oct/25)
TODO: Play Piano (2020/Oct/22) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/24) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/26) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/28) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Oct/30) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Nov/1) every 2 days until 2020/Nov/4
TODO: Play Piano (2020/Nov/3) every 2 days until 2020/Nov/4
DONE: Play Piano
→  lesson03
```



Take Home Message

POLYMORPHISM is one of the four core concepts of OOP

Overloaded functions / operators implemented as member, free 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

Next Lectures

~~Lesson 00: Introduction~~

~~Lesson 01: Hello, world!~~

~~Lesson 02: User-defined types~~

~~Lesson 03: Inheritance~~

~~Lesson 04: Polymorphism~~

Lesson 05: STL Containers

Lesson 06: Indirection

Lesson 07: Templates

Lesson 08: Exceptions

So let's go on C++ training with ❤️

(c) 2020/2021 - dominique.ginhac@u-bourgogne.fr - @dginhac