Lecture 7: Objects and classes in C++

1 Object-oriented programming

C++ is an *object-oriented programming* (OOP) language. One of the key advantages of OOP is that it allows you to structure your program by dividing functionality into smaller, self-contained *objects*. This means you can keep your main code simple and readable while encapsulating related data and behaviour in separate *classes*, making programs easier to manage, maintain, and extend.

1.1 Objects and classes

You have already encountered *objects* and *classes* in this course, but so far you have not defined any yourselves. Since the beginning of the course, you have been building a functions package, `myfunctions.cxx`, along with its corresponding header file, `myfunctions.h`, which contains the necessary function declarations and constructors.

We now want to extend this framework by incorporating user-defined *classes* and *objects*. Remember that instances of a *class* — i.e., variables belonging to that class — are called *objects*. In this example, we construct a scenario that requires functions and objects to interact, so the previously mentioned *include guards* are essential to prevent multiple inclusions of the header file. Let's consider the following example, where we define two simple `class` types, `operatingsystem` and `car`, starting with their shared header file, `myobjects.h`:

```
1  // myobjects.h
2  #ifndef MYOBJECTS_H
3  #define MYOBJECTS_H
4
5  #include <string>
6  #include "myfunctions.h"
7  using namespace std;
8  class operatingsystem{
9   private:
10   // member variables
11   string name;
12   int year_of_publication;
13  public:
14   // methods
15   void set_name(string _name);
16   string get_name();
17   void set_year_of_publication(int _year_of_publication);
18   int get_year_of_publication();
19   void sayhelloto();
20 };
21
22 class car{
23  private:
24   // member variables
25   string name;
26   string model;
27   int year;
28  public:
29   // constructor
30   car(){}
```

```
  car(string _name, string _model, int _year);
  // destructor
  ~car(){}
  // methods
  void set_name(string _name);
  string get_name();
};

#endif // MYOBJECTS_H
```

Note that the header file defines the structure of each class, including the variables it contains. In compound variables like the objects above, you can include as many member variables as needed. Keep in mind, however, that increasing the number of member variables also increases the memory footprint of each object. The following *access specifiers* (also called *access modifiers*) can be used in class headers to define from where *member variables* and *methods* can be accessed or modified:

```
public:     // accessible from anywhere
private:    // accessible only from within the class itself
protected:  // accessible only from within the class itself and its derived (child) classes
```

To ensure modularity and maintainability of the code, it is good practice to keep member variables as *private* as possible. The `protected` and `private` access specifiers behave in the same way as long as no derived classes are involved.

The following listing shows the implementation file `myobjects.cxx`, which contains the definitions of the member functions declared in the corresponding header file:

```
// myobjects.cxx
#include <iostream>
#include "myobjects.h"
using namespace std;
// methods for operatingsystem
void operatingsystem::set_name(string _name){
  name = _name;
};
string operatingsystem::get_name(){
  return name;
};
void operatingsystem::set_year_of_publication(int _year_of_publication){
  year_of_publication = _year_of_publication;
};
int operatingsystem::get_year_of_publication(){
  return year_of_publication;
};
void operatingsystem::sayhelloto(){
  ::sayhelloto(name); // note the :: to indicate global scope
};

// constructor for car
car::car(string _name, string _model, int _year){
  name = _name;
  model = _model;
  year = _year;
}
// methods for car
void car::set_name(string _name){
  name = _name;
};
string car::get_name(){
  return name;
};
```

We have linked the user-defined class `operatingsystem` with our functions package by calling the function `::sayhelloto(name)` declared in `myfunctions.h` from within the method `void operatingsystem::sayhelloto()`. Here, the `::` refers to the *global namespace* and is necessary to avoid a naming conflict with the *member function* `void car::sayhelloto()`, which would otherwise have the same name.

The following is a simplified version of `myfunctions.h`, which also introduces a reverse dependency: the function void `sayhelloto(operatingsystem & myos)` requires that the class `operatingsystem` be defined beforehand.

```
1  // myfunctions.h
2  #ifndef MYFUNCTIONS_H
3  #define MYFUNCTIONS_H
4
5  #include <iostream>
6  #include <string>
7  #include "myobjects.h"
8  using namespace std;
9  class operatingsystem;
10 void sayhelloto(string input);
11 void sayhelloto(operatingsystem & myos);
12
13 #endif // MYFUNCTIONS_H
```

The *forward declaration* `class operatingsystem;` on line 9 serves to resolve these circular dependencies.

The corresponding source file `myfunctions.cxx` contains only the functions necessary for this example:

```
1  // myfunctions.cxx
2  #include <iostream>
3  #include "myfunctions.h"
4  using namespace std;
5  void sayhelloto(string input){
6    cout << "Hello " << input << endl;
7  }
8  void sayhelloto(operatingsystem & myos){
9    cout << "Hello " << myos.get_name() << endl;
10 }
```

Finally, let's look at an example `main.cxx` program:

```
1  // main.cxx
2  #include <iostream>
3  #include "myfunctions.h"
4  #include "myobjects.h"
5  #include <ctime>
6  int year(){
7    time_t now = time(0);
8    tm *lt = localtime(&now);
9    int result = 1900 + lt->tm_year;
10   return result;
11 }
12 int main(int argc, char *argv[]){
13   operatingsystem a;
14   a.set_name("Mac OS X");
15   cout << a.get_name() << endl;
16   a.set_name("Windows 10");
17   cout << a.get_name() << endl;
18   a.set_year_of_publication(2015);
19   cout << a.get_name() << " was first released in " << a.get_year_of_publication() << " and
     ↪  is thus " << year() - a.get_year_of_publication() << " years old." << endl;
```

```
  sayhelloto(a);
  a.sayhelloto();
  operatingsystem Linux[5];
  Linux[0].set_name("Open SUSE");
  Linux[1].set_name("CentOS 7");
  Linux[2].set_name("Ubuntu");
  Linux[3].set_name("AlmaLinux 9");
  Linux[4].set_name("Linux Mint");
  for (int i = 0; i < 5; i++){
    cout << "Linux[" << i << "] = " << Linux[i].get_name() << endl;
  }
  car c;
  c.set_name("Mercedes");
  cout << c.get_name() << endl;
  car * test_car;
  test_car = new car("BMW", "X3", 2003);
  cout << test_car->get_name() << endl;
  delete test_car;
  return 0;
}
```

This sample code produces the following output:

```
Mac OS X
Windows 10
Windows 10 was first released in 2015 and is thus 10 years old.
Hello Windows 10
Hello Windows 10
Linux[0] = Open SUSE
Linux[1] = CentOS 7
Linux[2] = Ubuntu
Linux[3] = AlmaLinux 9
Linux[4] = Linux Mint
Mercedes
BMW
```

If you have been properly compiling your functions package so far, all you need to do now is add an extra step to your `Makefile` to compile `myobjects.cxx` into `myobjects.o`, and then link `myobjects.o`, `myfunctions.o`, and `main.cxx` together.

Below is an example `Makefile`, similar to the one you should have already created and maintained in the practical sessions. This version has been extended to include the compilation of the new object-oriented parts (`myobjects.o`, etc.):

```
#This is the directory of YOUR source code
sourcedirectory=./
#These are your source codes and components
myfunctions=myfunctions.cxx
myobjects=myobjects.cxx
mymain=main.cxx
output=$(mymain:.cxx=)
# Here, we're defining the compilers
CC=gcc
CPP=g++
NVCC=nvcc
# We're defining systems variable such as "remove" from system and "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M_%S" )
SFLAG=-Wall
# When a Makefile is executed, by default it tries the option "all"
all: clean functions objects main
# We tell the makefile to clean, then to compile the functions and objects, and eventually
↪  to compile main and to link it with the other components.
functions:
```

```
# Compile the functions part
	@echo Compiling $(sourcedirectory)$(myfunctions:.cxx=.o)
	$(CPP) -c -o $(myfunctions:.cxx=.o) $(SFLAG) $(sourcedirectory)$(myfunctions)
	@echo Successfully compiled $(sourcedirectory)$(myfunctions)
	@echo The unlinked compiled code is $(myfunctions:.cxx=.o)
objects:
# Compile the objects part
	@echo Compiling $(sourcedirectory)$(myobjects)
	$(CPP) -c -o $(myobjects:.cxx=.o) $(SFLAG) $(sourcedirectory)$(myobjects)
	@echo Successfully compiled $(sourcedirectory)$(myobjects)
	@echo The unlinked compiled code is $(myobjects:.cxx=.o)
main:
# Compile main and link it with functions and objects
	@echo Linking $(myobjects:.cxx=.o) and $(myfunctions:.cxx=.o) with $(mymain)
	$(CPP) -o $(addprefix run_,$(output).exe) $(SFLAG)
	↪ $(sourcedirectory)$(myobjects:.cxx=.o) $(sourcedirectory)$(myfunctions:.cxx=.o)
	↪ $(mymain)
	@echo Successfully compiled $(sourcedirectory)$(output)
	@echo Everything is linked and compiled into $(addprefix run_,$(output).exe)
clean:
	@echo $(TIMESTAMP)
	@echo "Making old/$(TIMESTAMP) directory"
	$(shell mkdir -p old/$(TIMESTAMP) )
	@echo "Copying the source to the old directory"
	$(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
	$(shell cp -r $(sourcedirectory)/*.h old/$(TIMESTAMP) )
	@echo "Moving all .exe to the old directory"
	$(shell mv *.exe old/$(TIMESTAMP) )
	$(shell mv *.o old/$(TIMESTAMP) )
	@echo "Copying the Makefile to the old directory"
	$(shell cp Makefile old/$(TIMESTAMP) )
```

1.2 Operator Overloading

It is in general very useful to be able to define operators like `+`, `-`, `*`, etc. for mathematical operations, or `<<` for stream operations for our own classes as well. This becomes possible through the concept of *operator overloading*.

When overloading an operator, we define a function that specifies how the operator behaves when it is used with objects of a particular class. As an example, consider overloading the `<<` operator for our car class. Note that, in this case, we are not declaring a method of our new class. Instead, we define a separate function that is called when an `ostream` and an object of our class `car` are used with the `<<` operator:

```
friend ostream & operator << (ostream & output, const car & c);
```

The `friend` keyword allows this function to access the `private` variables and methods of the class `car`. The declaration therefore needs to be included in the `class` definition.
The corresponding function body can then be defined as:

```
ostream & operator << (ostream & output, const car & c){
  output << c.name << ", " << c.model << " (" << c.year << ")";
  return output;
}
```

One could also define the function *without* the `friend` keyword — consequently, the declaration would not be part of the function definition — but in that case only `public` members and methods would be available.

Operator overloading can also be useful for classes representing mathematical objects. As a simple example, we define a class representing a two-dimensional vector with components of type `double`. In addition to ordinary member functions, we can overload operators to define, for example, vector negation, vector addition, and the scalar product of two vectors:

```
// vector2d.cxx
#include <iostream>
using namespace std;
class vector2d{
private:
    double x;
    double y;
public:
  // constructor
  vector2d(double _x, double _y){ x = _x; y = _y; }
  // getters
  double get_x() const { return x; }
  double get_y() const { return y; }
  // operators
  vector2d operator - () const { return vector2d(-x, -y); }
  vector2d operator + (const vector2d & v) const { return vector2d(x + v.x, y + v.y); }
  double operator * (const vector2d & v) const { return x * v.x + y * v.y; }
  //  friend ostream & operator << (ostream & output, const vector2d & v){ output << "(" <<
  ↪  v.x << ", " << v.y << ")"; return output; }
};
ostream & operator << (ostream & output, const vector2d & v){ output << "(" << v.get_x() <<
↪  ", " << v.get_y() << ")"; return output; }

int main(){
  vector2d a(1., 2.);
  vector2d b(3., 4.);
  cout << "-a = " << -a << endl;
  cout << "a.operator - () = " << a.operator - () << endl;
  cout << "a + b = " << a + b << endl;
  cout << "a.operator + (b) = " << a.operator + (b) << endl;
  cout << "a * b = " << a * b << endl;
  cout << "a.operator * (b) = " << a.operator * (b) << endl;
  return 0;
}
```

In the `main` function, the respective second output lines illustrate that overloaded operators follow the same basic syntax as ordinary functions and methods: `operator` followed by the operator symbol forms the function name. At the same time, operators provide a convenient shorthand notation. *Binary operators* can be written with the operator between the two operands, while *unary operators* can be written directly in front of the operand.

For overloading the `<<` operator, we use the version without the `friend` keyword here. The function accesses the `vector2d` object only through its `public` member functions. Alternatively, the operator could be declared as a `friend` function inside the `class`, which would allow it to access `private` members directly while providing the same functionality.

The `const` qualifier at the end of a member function guarantees that the object on which the function is called is not modified. The `const` qualifier of a reference parameter guarantees that the referenced argument is not modified. In addition, a `const` reference can bind to temporary objects, such as the result of another operation.

Let's now practice creating and working with some classes and objects.

We will now start a 60-minute practical programming session.

2 Practical Session – Part I

Objects and classes - Extend your Functions by an Objects Package

1. Create the above objects `car` and `operatingsystem` as introduced in the lecture — in new files `myobjects.cxx` and `myobjects.h`. Extend also your `Makefile` to deal with them.
2. Create a *getter* and a *setter* function for each member variable of the two classes.
3. Create methods for both classes to tell you how old each object is.
4. Create an `array` of `operatingsystem` objects and set the properties for at least 5 operating systems.
5. Create a `vector` of `car` objects and set the properties for at least 5 cars.
6. Repeat exercises 4 and 5 using a dynamically allocated `array` and a `vector`, respectively.
7. Create a `threevector` class for 3-dim. vectors.
   (a) Declare and define *private* member variables to hold the three components, and *getter* and *setter* methods to each of them;
   (b) Declare and define constructors to create a `threevector` from three `double`s, from an `array` of `double`s, a `vector` of `double`s, and from a `threevector`;
   (c) Declare and define methods to set all three components in one call, and to calculate the (squared) length of a `threevector`;
   (d) *Overload* the operators `+`, `-`, to add and subtract two `threevector`s, as well as `*`, `/` to define inner and outer products.
   (e) *Overload* the operator `<<` for the class `threevector` so that objects of this class can be printed directly to an `ostream` (e.g., `cout`).
8. Create a `matrix3x3` class for 3×3 matrices.
   (a) Declare and define *private* member variables, e.g. as a nested `vector` or `array`.
   (b) Declare and define *getter* and *setter* methods, for the full matrix and for individual components.
   (c) Declare and define a method to calculate the determinant of a `matrix3x3`.
   (d) *Overload* the operators `+`, `-`, `*` to add, subtract and multiply two `matrix3x3`s. Further *overload* `*` to define scalar multiplication and the multiplication of a `matrix3x3` and a `threevector`, which results in a `threevector`.
   (e) *Overload* the operator `<<` for the class `matrix3x3` so that objects of this class can be printed directly to an `ostream` (e.g., `cout`).

We will now continue with a 15-minute theory session.

3 Inheritence, derived classes and polymorphism

In this part of the lecture, we go beyond simple classes and learn how to define *derived classes* that inherit from one or more *base classes*. For simplicity, we will use the `car` class as an example and extend it by defining two derived classes: `electric_car` and `gas_car`. These derived classes share common features (member variables and methods) with the base class, but can also have additional individual features on their own.

The following shows an example of a common header file `myobjects.h` for these classes:

```
// myobjects.h
#ifndef MYOBJECTS_H
#define MYOBJECTS_H

#include <string>
using namespace std;
class car{
 protected:
  // member variables
  string name;
  string model;
  int year;
 public:
  // constructors
  car(){}
  car(string _name, string _model, int _year);
  // destructor
  virtual ~car(){}
  // methods
  void set_name(string _name);
  string get_name();
  virtual void drive();
};

class electric_car : public car {
private:
  int battery_capacity; // in kWh
public:
  // constructor
  electric_car(string _name, string _model, int _year, int _battery_capacity);
  // destructor
  ~electric_car() override {}
  // methods
  void drive() override;
  void capacity_info();
};

class gas_car : public car {
private:
  int tank_size; // in liters
public:
  // constructor
  gas_car(string _name, string _model, int _year, int _tank_size);
  // destructor
  ~gas_car() override {}
  // methods
  void drive() override;
  void tank_info();
};

#endif // MYOBJECTS_H
```

Some features in the file above are particularly important for understanding base classes and derived classes:

- *class declaration with inheritance:* Derived classes inherit the accessible members of their base class(es). This allows derived classes to build upon and extend the functionality of their base classes.
- *Constructors and destructors:* Constructors of base classes are called before the derived class constructor runs, and destructors are called in the reverse order. This ensures proper initialization and cleanup of both base and derived parts of an object.
- *Method overriding:* Methods in a derived class can replace or extend the behaviour of base-class methods. The `override` keyword (optional but highly recommended) makes this explicit and allows the compiler to catch errors if the base method signature changes.
- *Virtual methods and polymorphism:* By declaring a method as `virtual` in the base class, we enable dynamic dispatch. When the method is called through a base-class pointer or reference, the implementation corresponding to the actual object type is selected at runtime.

The corresponding source code of `myobjects.cxx` looks as follows:

```
1  // myobjects.cxx
2  #include <iostream>
3  #include "myobjects.h"
4  using namespace std;
5  // constructor for car
6  car::car(string _name, string _model, int _year){
7    name = _name;
8    model = _model;
9    year = _year;
10 }
11 // methods for car
12 void car::set_name(string _name){
13   name = _name;
14 };
15 string car::get_name(){
16   return name;
17 };
18 void car::drive(){
19   cout << "The car " << name << " is driving." << endl;
20 };
21
22 // constructor for electric_car
23 electric_car::electric_car(string _name, string _model, int _year, int _battery_capacity) :
   ↪  car(_name, _model, _year){
24   battery_capacity = _battery_capacity;
25 }
26 // methods for electric_car
27 void electric_car::drive(){
28   cout << name << " " << model << " (electric) is silently gliding down the road. " << endl;
29 }
30 void electric_car::capacity_info(){
31   cout << "Battery capacity: " << battery_capacity << " kWh." << endl;
32 }
33
34 // constructor for gas_car
35 gas_car::gas_car(string _name, string _model, int _year, int _tank_size) : car(_name,
   ↪  _model, _year){
36   tank_size = _tank_size;
37 }
38 // methods for gas_car
39 void gas_car::drive(){
40   cout << name << " " << model << " (gas) is roaring down the highway. " << endl;
41 }
42 void gas_car::tank_info(){
43   cout << "Tank size: " << tank_size << " liters." << endl;
44 }
```

The derived classes choose which base constructor to use via a *constructor initializer list*. In our example, this is done in the derived-class constructors with `: car(_name, _model, _year)` . This syntax ensures that the `car` base class is properly constructed with the correct initialization parameters before the derived class adds its own initialization.

The definition of methods in derived classes, both overridden and newly added methods, works in the same way as for methods of the base class. When overriding a method, it is still possible to explicitly call the base-class implementation by using the scope resolution operator, for example: `car::drive();`. This is useful when the derived method is intended to extend rather than completely replace the behaviour of the base-class method.

A sample `main.cxx` code to test these features is:

```
// main.cxx
#include <iostream>
#include <vector>
#include "myobjects.h"
using namespace std;
int main(){
  vector<car*> cars(5);
  cars[0] = new electric_car("Tesla", "Model S", 2025, 100);
  cars[1] = new gas_car("Ford", "Mustang", 2020, 60);
  cars[2] = new electric_car("Nissan", "Leaf", 2011, 40);
  cars[3] = new gas_car("Toyota", "Corolla", 1966, 50);
  cars[4] = new electric_car("BMW", "i3", 2013, 42);

  cout << "Driving cars" << endl << endl;
  for (size_t i = 0; i < cars.size(); i++){
    cars[i]->drive();
  }
  cout << endl;

  cout << "Cleaning up memory" << endl << endl;
  for (size_t i = 0; i < cars.size(); i++){
    delete cars[i];
  }

  return 0;
}
```

Polymorphism in `C++` lets you treat objects of different derived types in the same way through a pointer or reference to a base class. Even though the `cars vector` is declared as `vector<car*>`, each element actually points to a different type of `car` (`electric_car`, `gas_car`, etc.). When you call `cars[i]->drive();` , `C++` looks at the actual object type at runtime (not the pointer type) and calls the correct `drive()` method for that specific `car`. This behaviour is only possible because `drive()` is declared as `virtual` in the base class.

This sample code produces the following output:

```
--- Driving cars ---
Tesla Model S (electric) is silently gliding down the road.
Ford Mustang (gas) is roaring down the highway.
Nissan Leaf (electric) is silently gliding down the road.
Toyota Corolla (gas) is roaring down the highway.
BMW i3 (electric) is silently gliding down the road.
--- Cleaning up memory ---
```

Note that we have removed the (artifical) dependence on `myfunctions.h` and `myfunctions.cxx` for simplicity. A possible `Makefile` that no longer depends on them is:

```
#This is the directory of YOUR source code
sourcedirectory=./
#These are your source codes and components
myobjects=myobjects.cxx
mymain=main.cxx
output=$(mymain:.cxx=)
# Here, we're defining the compilers
CC=gcc
CPP=g++
NVCC=nvcc
# We're defining systems variable such as "remove" from system and "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M_%S" )
SFLAG=-Wall
# When a Makefile is executed, by default it tries the option "all"
all: clean objects main
# We tell the makefile to clean, then to compile the objects, and eventually to compile main
↪  and to link it with the other components.
objects:
# Compile the objects part
        @echo Compiling $(sourcedirectory)$(myobjects)
        $(CPP) -c -o $(myobjects:.cxx=.o) $(SFLAG) $(sourcedirectory)$(myobjects)
        @echo Successfully compiled $(sourcedirectory)$(myobjects)
        @echo The unlinked compiled code is $(myobjects:.cxx=.o)
main:
# Compile main and link it with objects
        @echo Linking $(myobjects:.cxx=.o) with $(mymain)
        $(CPP) -o $(addprefix run_,$(output).exe) $(SFLAG)
        ↪  $(sourcedirectory)$(myobjects:.cxx=.o) $(mymain)
        @echo Successfully compiled $(sourcedirectory)$(output)
        @echo Everything is linked and compiled into $(addprefix run_,$(output).exe)
clean:
        @echo $(TIMESTAMP)
        @echo "Making old/$(TIMESTAMP) directory"
        $(shell mkdir -p old/$(TIMESTAMP) )
        @echo "Copying the source to the old directory"
        $(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
        $(shell cp -r $(sourcedirectory)/*.h old/$(TIMESTAMP) )
        @echo "Moving all .exe to the old directory"
        $(shell mv *.exe old/$(TIMESTAMP) )
        $(shell mv *.o old/$(TIMESTAMP) )
        @echo "Copying the Makefile to the old directory"
        $(shell cp Makefile old/$(TIMESTAMP) )
```

3.1 Multiple inheritance

C++ does not restrict a class to inherit from only one base class; a class can be derived from multiple base classes. Building on our existing `car`, `electric_car`, and `gas_car` classes, we can create a new `hybrid_car` class that inherits from both `electric_car` and `gas_car`. This demonstrates how multiple inheritance allows us to combine features from two (or more) existing classes without fundamentally changing the original class hierarchy. The new functionality is introduced simply by adding a new class declaration — no need to modify the existing `car`, `electric_car`, or `gas_car` definitions:

```
class hybrid_car : public electric_car, public gas_car {
public:
  // constructor
  hybrid_car(string _name, string _model, int _year, int _battery_capacity, int _tank_size)
  ↪  : car(_name, _model, _year), electric_car(_name, _model, _year, _battery_capacity),
  ↪  gas_car(_name, _model, _year, _tank_size) {}
  // method for hybrid_car
  void drive() override;
};
```

We actually do need to modify a few lines in the declarations of the classes that `hybrid_car` inherits from. However, these modifications are *minimal*: we only *prepare* these classes for use as base classes, without changing their functionality. Specifically, we replace the relevant lines in `myobjects.h` with:

```
25 class electric_car : virtual public car {
26 protected:
```

and

```
38 class gas_car : virtual public car {
39 protected:
```

The keyword `virtual` has been inserted to mark the inheritance as *virtual*. This tells `C++` to share a single copy of the `car` base class whenever multiple inheritance paths lead back to it.

This matters because `electric_car` and `gas_car` both inherit from `car`. When `hybrid_car` inherits from both of them, two inheritance paths lead to car:

- *Without virtual inheritance*: `hybrid_car` would contain two independent `car` subobjects, causing ambiguity (which `car::` method do you call?) and wasting memory.
- *With virtual inheritance*: All paths share one unified `car` subobject, solving the well-known *diamond problem* in `C++`.

Even if `electric_car` or `gas_car` are not currently intended to serve as base classes for further derived classes, virtual inheritance can be used to prepare the hierarchy for possible future extensions. This can be useful if the class hierarchy may later be extended in a way that would otherwise introduce multiple copies of the same base-class subobject.

Replacing `private` with `protected` is unrelated to *multiple inheritance*. It simply ensures that the members of the respective base class remain accessible in derived classes — in this case, that the members of `electric_car` or `gas_car` are accessible in `hybrid_car`. Using `protected` instead of `private` does not change the behaviour of `electric_car` or `gas_car` themselves. It only affects how other classes can derive from them and access their members.

In `myobjects.cxx`, we only need to *override* the `drive()` method for `hybrid_car`:

```
46 // method for hybrid_car
47 void hybrid_car::drive() {
48   cout << name << " " << model << " can drive using a " << battery_capacity << " kWh battery
   ↪  or a " << tank_size << " l fuel tank." << endl;
49 }
```

With the `hybrid_car` class set up this way, no further modifications are needed to use polymorphism. We can simply add `hybrid_car` objects to our `vector<car*> cars`, just like we do with `electric_car` or `gas_car`:

```
13   cars.push_back(new hybrid_car("Toyota", "Prius", 1997, 8, 45));
```

And the additional line of output comes from the `drive()` method of `hybrid_car`:

```
Toyota Prius can drive using a 8 kWh battery or a 45 L fuel tank.
```

We will now start a 60-minute practical programming session.

4 Practical Session – Part II

Bringing It All Together - Extend your Functions and Objects Packages

1. If you haven't done so yet, create general functions that take an `array` of integers, a *stack* `vector<int>` or a *heap* `vector<int>`, and calculate max, min, median, average, sample skewness and sample standard deviation.

2. Write a class `data` that holds these statistical values as member variables and contains a method to output them.

3. Use `data` as a base class, and derive four classes that can also read in and store the data itself (from an `array` of integers of a given length) in the format of a *heap* and *stack* `array` and `vector`, respectively. Add methods to calculate the statistical values.

4. Create such an `array` with 10000 random integers as a sample *dataset*. Test the functionality of your derived classes to calculate and output the statistical properties (i.e. average, max, min, etc.) of this dataset.

We will now conclude with a 15-minute question session.

5 Conclusion

What you practiced in the last practical sessions reflects, in essence, how the `ROOT` analysis framework was developed at CERN. `ROOT` is a powerful framework featuring a large number of objects and built-in functions, along with a command-line interface that allows you to quickly test and prototype code without compiling it. In the next class, we will explore how to use `ROOT` 's scripting capabilities and command-line functionality in more detail.
Make sure that you have access to a working `ROOT` installation — either locally or through access to the physics institute's cluster.