Dynamic Polymorphism without Inheritance: Implications for Education

Ivaylo Donchev, Emilia Todorova
Department of Information Technologies
St Cyril and St Methodius University of Veliko Tarnovo
Veliko Tarnovo, Bulgaria

***Abstract*—Polymorphism is a core OO concept. Despite the rich pedagogical experience in teaching it, there are still difficulties in its correct and multifaceted perception by students. In this article, a method about a deeper study of the concept of polymorphism is offered by extending the learning content of the CS2 C++ Programming course with an implementation variant of dynamic polymorphism by type erasure, without using inheritance. The research is based on an inductive approach with a gradual expansion of functionalities when introducing new concepts. The stages of development of such a project and the details of the implementation of each functionality are traced. The results of experimental training showed higher scores of the experimental group in mastering the topics related to polymorphism. Based on these findings, recommendations for the construction of the lecture course and the organization of the laboratory work are suggested.**

***Keywords*—Inheritance; polymorphism; object-oriented; C++; type erasure; pointers; templates; lambda expressions; teaching**

I. Introduction

The study of the concept of polymorphism is widely used in programming courses. Students encounter its forms already in the introductory course and use them successfully, even if they do not perceive them as such at first. This is where function overloading and type casting (coercion polymorphism) come in. Later, parametric polymorphism (templates, generics) is added. However, when one says polymorphism, without specifying what kind, one usually means the inherent OOP inclusion (subtype) polymorphism, implemented through inheritance, virtual methods and dynamic linking (dynamic polymorphism). Subtype polymorphism is a cornerstone of object-oriented programming. By hiding variability in behavior behind a uniform interface, polymorphism decouples clients from providers and thus enables genericity, modularity, and extensibility [1]. This type of polymorphism, together with parametric, forms the more general universal polymorphism category of the popular classification of polymorphism types given in [2], and overloading and coercion form the ad-hoc category.

Bjarne Stroustrup defines polymorphism as "providing a single interface to entities of different types" [3] and distinguishes between dynamic (run-time) polymorphism, implemented through virtual functions through an interface provided by a base class, and static (compile-time) polymorphism through overloaded functions and templates. It is considered useful to differentiate between the two types. The focus is on three matters: time when the selection of the specific method occurs (run-time or compile-time); different behavior, based on dynamic or static type; means by which it is usually achieved – inheritance in case of dynamic and overloading and templates in case of static.

Virtual functions and inheritance are typical means of achieving dynamic polymorphism, but they have their drawbacks in terms of performance and flexibility. The authors believe that in order to build deep understanding of the concept of dynamic polymorphism, students should have an idea that it can be achieved by other means, such as type erasure. Such an implementation with the currently available capabilities of C++ is cumbersome and error-prone, requires the definition of many types and functions, and provides no distinguishable advantages over inheritance, but has a beneficial impact on developing the programming competencies of computer science students, especially regarding the proper use of pointers.

In this paper, experience of implementing dynamic polymorphism without inheritance is shared (with manual implementation of virtual tables and the implementation of copy and move semantics) within the elective course "Programming in C++" for students majoring in Software Engineering. Their curiosity and previous experience with C# provoked to try to implement a familiar sample project in a new way. The project evolves incrementally with the addition of new functionalities as the relevant concepts are studied. The audience is later expanded to include Computer Science students with no **.**NET experience in the mandatory C++ Programming course. The difficulties which students encounter are explored and the effect of deeper study of concepts is tracked.

II. Methodology

A. Motivation

A classic example of inheritance polymorphism that our students have covered in the C# course is the hierarchy shown in Fig. 1. The base class declares a virtual method `Accelerate()`, which the two derived classes override. The classes can be used as follows:

```
Vehicle vehicle = new Car();
vehicle.Accelerate();
vehicle = new Truck();
vehicle.Accelerate();
List<Vehicle> vehicles = new() {
```

```
 new Car(), new Truck(), new Truck(), new Car()
};
foreach (var v in vehicles){
   v.Accelerate();}
```

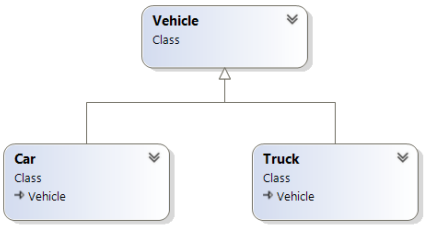

Fig. 1. The Hierarchy that will be Implemented.

With no major changes to the client code, by implementing an interface `IVehicle`, declaring the method `Accelerate()`, the same functionality without inheritance can be obtained. The C++ hierarchy implementation is even shorter – an abstract base class with pure virtual method `Accelerate()`, which is inherited and overridden by the classes `Car` and `Truck`. However, the client code requires use of pointers (dynamic allocation – one of the problems with inheritance), not objects, because abstract class cannot be instantiated:

```
Vehicle *vehicle = new Car{};
vehicle->Accelerate();
delete vehicle;
vehicle = new Truck{};
vehicle->Accelerate();
std::vector<Vehicle*> vehicles{
  new Car{},
  new Truck{},
  new Truck{},
  new Car{}
};
for (auto&& v : vehicles){
  v->Accelerate();
}
```

However, students want to write the code as they are used to with C#:

```
Vehicle v = Car{};
v.accelerate();
v = Truck{};
v.accelerate();
std::vector<Vehicle> vehicles{
 Car{}, Truck{}, Truck{}, Car{} };
for (auto&& v : vehicles) {
  v.accelerate();
}
```

If a default implementation of the method in the class `Vehicle` is added, it is no longer abstract, this code will be compiled, but there will be not polymorphic behavior – the same method will always be executed – that of the class `Vehicle`. For this code to work correctly, it is needed to implement manually the virtual functions mechanism, which will be done in the following sections.

B. Inheritance Problems

To prove why it is necessary to look for inheritance-free design possibilities, it is needed to look at some of the problems it raises. In his talk at CppCon 2020, Simon Brand summarizes and analyzes five issues related to inheritance [4]:

- Often requires dynamic allocation

This problem is encountered when implementing the hierarchy of Fig. 1. The attempt to store in `vector<Vehicle>` objects of the derived classes `Car` and `Truck` leads to what's called "slicing" – just the inherited from `Vehicle` (base) part of the object are got, and we slice of the dynamic part of the derived object. Usually this is not what is needed. The same is the situation when a function returns an object of the base class by value. To avoid this problem, it is needed to allocate and return a pointer dynamically (in the case of the function), or store a pointer (raw or smart) in a vector:

```
std::vector<std::unique_ptr<Vehicle>> vehicles;
```

- Ownership and nullability considerations

When working with pointers their ownership and validity must always be considered. If `unique_ptr` is used, then the ownership is clear. It is not so clear, however, if a function is returning `unique_ptr`. Questions arise: can it return null? Is it necessary to check? If the function accepts a `unique_ptr` argument, what happens if null is passed? Is that valid? What's the behavior? Too many questions to take care of.

- Intrusive: requires modifying child classes

Supporting inheritance requires modifying child classes.

```
namespace LibOne {
        class Base {
        public:
          virtual void Foo();
        };
}
namespace LibTwo {
        class Other {
        public:
          virtual void Foo();
        };
}
```

There is a Base class in LibOne and then there is some other library LibTwo which has an Other class. They both have Foo() method which returns void and they're virtual. It is not possible to allocate an instance of LibTwo::Other and take a pointer to it through a LibOne::Base.

```
LibOne::Base* b = new LibTwo::Other{};
```

This will not work because LibTwo::Other does not say it inherits from LibTwo::Other. This can be a problem. Maybe LibTwo::Other cannot be changed like if we could just decorate it to inherit from Base. For example, the code may not be available or there may be other restrictions that prevent from doing so. So, polymorphism with inheritance is intrusive.

- No more value semantics

Again, the question comes to the pointers. If we want value semantics, then something on top must be built. For example, virtual Clone() function which uses the correct dynamic type, dynamically allocate a pointer, and pass it back. That's a way of getting a copy behavior but still it is not the usual C++ value semantics which a lot of code depends on.

- Changes semantics for algorithms and containers

Inheritance changes semantics for algorithms and containers. If a std::sort() is done, maybe sorting on pointers is on, and custom comparator object must be supplied. The same situation is when these things are stored in a std::set. Another situation that must be thought about is that not usual C++ values are used, but it is desirable for most of the C++ development to use value semantics.

C. Implementing Virtual Functions by Hand

The start is with an implementation of the hierarchy of Fig. 1 in the traditional way, by inheritance.

```
class Vehicle {
public:
        virtual ~Vehicle();
        virtual void Accelerate() const = 0;
};
class Car : public Vehicle {
        virtual void Accelerate() const override;
};
class Truck : public Vehicle {
        virtual void Accelerate() const override;
};
```

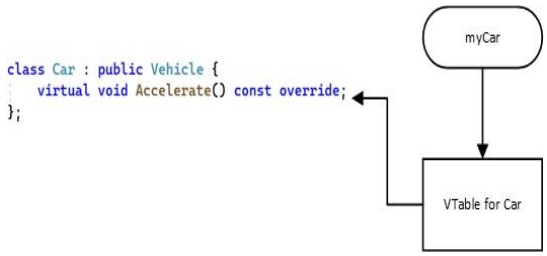

Fig. 2. Situation with an Object of the Car Class.

It is important for students to understand how virtual functions works internally. In Fig. 2 myCar is an object of the Car class. Then it is going to have a pointer to a VTable – a virtual table. This table takes care of how to call virtual functions in a polymorphic object. VTable in turn has a pointer to the Accelerate() function for Car. So, there is a couple of indirections that are gone through when Accelerate() is called. First, the VTable must be grabbed, then the VTable must be read through to get the Accelerate() function. That can be a performance bottleneck.

The interface to be implemented includes a Vehicle class, which should provide all the necessary functionality for polymorphic behavior. The other two classes Car and Truck just need to have accelerate() functions. These two classes should not inherit Vehicle, but there is a need to support this use case. It should be possible to create a Vehicle from a Car and call accelerate(). The aim is ability to create a Vehicle from a Truck and have it accelerate() and all this should be done without doing any slicing.

```
Vehicle c = Car{};
v.accelerate();
Vehicle t = Truck{};
t.accelerate();
```

To implement the virtual functions manually several steps have to be made:

- Declare virtual table for the abstract interface

First, it is declared what the virtual table layout looks like for `Vehicle` class.

```
struct VTable {
        void (*accelerate)(void* ptr);
        void (*destruct)(void* ptr);
};
```

VTable has two function pointers – one for accelerate and one for destruct, which will be called by the destructor of the specific object. And since memory within the object will be allocated, it's going to reclaim that memory. The arguments of the two function pointers are void pointers, because in this way the concrete object will be stored internally. Void pointers will be passed, and then the concrete objects are going to cast those pointers internally.

- Define virtual table for a concrete type

```
template<typename T>
VTable vtable_for {
 [](void* p) {
   static_cast<T*>(p)->accelerate(); },
 [](void* p) {
   delete static_cast<T*>(p);}
};
```

This is a variable template (available since C++14). There is an instance of a VTable and it's templated on the concrete type T, i.e., Car or Truck. So, a function which is going to call the correct version of accelerate() is needed, and a function which deletes the object and calls the destructor. Lambdas can be used for this purpose. For a given concrete object the first function pointer is just going to static cast to the concrete type and then call accelerate(). And then the second function is going to static cast and then calls delete. This all works because lambdas which don't capture can decay to function pointers.

- Capture the virtual table pointers on construction

When a Vehicle class is constructed, it is needed to fill in the pointer for the concrete object and pointer to the virtual table. The constructor will be implemented as a template that accepts any type as an argument. In a real situation, of course, it would be good to limit the possible types. Memory will be allocated dynamically for the object received as an argument and a copy of it in p_obj will be saved, and after that a pointer to our virtual table will be taken and stored inside vehicle (in the p_vtable field).

```
class Vehicle {
public:
        void* p_obj;
        VTable const* p_vtable;
        template<typename T>
        Vehicle(T const& obj) :
p_obj(new T(obj)),
p_vtable(&vtable_for<T>)
{}
  };
```

It is noted that since there is current access to what type the obj is (Car or Truck), that information is saved for later by grabbing the right VTable pointer and by dynamically allocating a copy of our obj. This technique is called “type erasure”.

- Forward calls through the virtual table

Finally, it is needed to forward the calls through the virtual table.

```
class Vehicle {
    //...
  void accelerate() {
    p_vtable->accelerate(p_obj);
  }
  ~Vehicle() {
    p_vtable->destruct(p_obj);
  }
};
```

Inside the Vehicle class if accelerate() is called, then we indirect through p_vtable and pass it the void pointer. And that is then going to cast inside the function and call the right version. And then similarly for the structure we call destruct().

So now the students have something which works for that use case. It remains to define the classes Car and Truck with the corresponding implementations of the function accelerate().

```
class Car {
public:
  void accelerate() {
    std::cout << "The car accelerates.\n";
  }
};
class Truck {
public:
  void accelerate() {
    std::cout << "The truck accelerates.\n";
   }
  };
```

It is possible to construct car; it is possible to construct vehicle from a car and a truck and make them accelerate and all it works. The goal set at the beginning of the section has been achieved.

D. Adding Copy and Move Semantics

So far, some of the inheritance problems discussed in section B nave been solved. There are no more problems with ownership and nullability, because now all memory allocations are handled inside the Vehicle class. There are no pointers externally. We're just dealing with the values. Intrusivity is avoided as well, because now Car and Truck don't inherit from anyone. However, the problem with value semantics remains, because these objects can't be copied or moved, but the VTable can be extended with a copy_() and a move_() function pointers and solve this problem.

```
struct VTable {
        //...
        void* (*copy_)(void* ptr);
        void* (*move_)(void* ptr);
};
```

The first function will allocate a copy, and the second will allocate by moving from the object. This functionality should be implemented by adding two new lambda functions to the variable template vtable_for.

```
template<typename T>
VTable vtable_for {
 //...
 [](void* p) -> void*{
   return new T(*static_cast<T*>(p)); },
 [](void* p) -> void*{
   return new T(std::move(*static_cast<T*>(p)));}
};
```

Now it is only needed in the copy constructor and move constructor of Vehicle to call from the virtual table p_vtable the corresponding functions.

```
Vehicle(Vehicle const& other) :
  p_obj(other.p_vtable->copy_(other.p_obj)),
  p_vtable(other.p_vtable)
{}
Vehicle(Vehicle&& other) noexcept :
  p_obj(other.p_vtable->move_(other.p_obj)),
  p_vtable(other.p_vtable)
{}
```

The same is done for copy assignment and move assignment operators.

```
Vehicle& operator=(Vehicle const& other) {
  p_obj = other.p_vtable->copy_(other.p_obj);
  p_vtable = other.p_vtable;
  return *this;
}
Vehicle& operator=(Vehicle&& other) noexcept {
  p_obj = other.p_vtable->move_(other.p_obj);
  p_vtable = other.p_vtable;
  return *this;
}
```

Now a much more complete interface is created, and a lot of actions can be performed – create vehicle from car and accelerate it; we can reassign it to a truck and make that accelerate. A new vehicle can be created from an old one. All works and there is value semantics even though we're doing dynamic polymorphism. It's just all handled under the covers. Students can even experiment by defining and traversing the vector of cars and trucks because we defined copying and moving.

```
Vehicle v = Car{};
v.accelerate();
v = Truck{};       // move assignment!
v.accelerate();
Vehicle t{ v };    // copy construction
t.accelerate();
std::vector<Vehicle> vehicles {
  Car{}, Truck{}, Truck{}, Car{} };
for (auto&& v : vehicles) {
v.accelerate();
  }
t = std::move(v);  // move assignment
```

Another inheritance problem is solved. There are already normal copy semantics and container semantics. The problem is that the code written to implement this functionality is too much and must be repeated for each class that needs to be handled dynamically. In addition, it's weird code and it's easy for something to go wrong.

III. Implications for Education

Teachers and students often consider learning programming a difficult pursuit [5]. Inheritance and polymorphism are arguably the most advanced and abstract subjects in object-oriented programming [6]. Their study involves many difficulties, which we will not consider here. The study in [7] identifies as the main cause of most problems the students' inability to understand what is happening with their program in memory, since they cannot build a clear mental model of the program's execution. Therefore, we believe that manually implementing the virtual tables will help overcome these issues. Our experience from the last two academic years shows progress in this direction. Students see that pointers are a very powerful tool and are motivated to study them. They look for literature and consult with the assistants, solve tasks independently. The result is a deeper understanding of memory management and program execution.

For students who have not studied C++ in the introductory course, after the topics related to the new-to-them syntax in terms of program structure, class definition, object instantiation, and message exchange, one should move on to learning about working with pointers and dynamic memory (something everyone else studied in the introductory C++ course). At an early stage, the topics of implementing the important OOP relations of composition and inheritance, which students know from the introductory course, should also be included. After that comes the time for an in-depth study of polymorphism. For students the goal is to learn to recognize polymorphism and model with it, not just to know its implementation in the specific language.

Dynamic binding and virtual functions should be seen as a mechanism in OOP languages to implement dynamic polymorphism, but not polymorphism to be considered as a consequence of using the mechanism. It is recommended to give a correct classification of the types of polymorphism in lectures in order to clarify the understanding of the concepts and distinguish them from the means of implementation.

A common mistake made by novice programmers is always to try to use the inheritance relationship. Undoubtedly, it is the most important and defining for the paradigm, but its application should not be overexposed in the course. After learning about the disadvantages of inheritance, students themselves will begin to look for alternative designs using other relations. In particular, it provoked interest for generic and functional programming.

Modern programming languages offer concepts from several programming paradigms. It is impossible to learn OOP in isolation from generic, functional, and procedural programming. Therefore, in parallel with OO concepts, other means like templates, lambdas, containers, and algorithms from STL should also go.

An important condition for successful training in OOP is the correct selection of the tasks that are considered in laboratory classes and given for homework. They should be such that polymorphism is a natural part of the solution. Suitable hierarchies to implement (both with and without inheritance) are as follows:

- base class Animal and derived classes Fish, Frog, Bird and polymorphic method Move(),
- base class Pet, derived classes Cat and Dog and polymorphic method MakeNoise(),
- base class Shape, derived Circle, Triangle, Rectangle and polymorphic methods Area() and Circumference(),
- basic class Publication, derived Magazine, Book and polymorphic method `Print()`.

Examples can be both from real life and more abstract. Examples with GUI components can also be used. It is a good idea to add a new class to an already implemented hierarchy, for example adding Motorcycle to Car and Truck. This can be

done as work to do by themselves within the lab exercise or given for homework.

Learning about abstract classes provides a good foundation to demonstrate the full power of polymorphism with perhaps its most typical application – the creation and manipulation of heterogeneous data structures. The vector of `Car` and `Truck` objects is such a structure as well. If there is enough time, a manual implementation of a heterogeneous singly linked list or binary tree can also be demonstrated in the training course.

Since the complete implementation of the Vehicle, Car, Truck hierarchy given in this paper covers many topics, it cannot be covered in a single lecture and practiced in a single lab session. It is recommended to teach incrementally, starting with an implementation with inheritance, and going through the type erasure option after discussing pointers, constructors, destructors, type conversion, lambda functions, function templates, variable templates. After the initial version, the project can be extended with an implementation of copy semantics only. This requires first familiarity with operator overloading and copy construction. Move semantics is not required for the classes to work correctly because it can be successfully replaced by copy. But since the choice of the C++ language in most cases is related to the increased requirements for speed and efficiency of code, it is recommended not to neglect it and to give and comment the described implementation of move semantics for the sample hierarchy. When learning the STL library, one can experiment with using different containers of Car and Truck objects and applying algorithms to them.

Since enough empirical data is not collected yet, the classic pedagogical experiment of proving the advantages of extended polymorphism learning is not completed. However, in Table I a comparison of the results for the academic year 2021-2022 is given of the start tests (in the beginning of the course) and the tests conducted immediately after the completion of the section devoted to polymorphism. The experimental group includes the students of the Software Engineering majors, for whom "Programming in C++" is an elective course in the 4th semester (12 people) and the Computer Science major, for whom the course is mandatory, but in the 2nd semester (35 people).

TABLE I. TEST RESULTS

Grade	Bachelor Program					
	SE		*CS*		*Informatics*	
	Start level	*End level*	*Start level*	*End level*	*Start level*	*End level*
A	25.0%	25.0%	14.3%	20.0%	9.1%	9.1%
B	16.7%	33.3%	22.9%	25.7%	27.3%	27.3%
C	33.3%	25.0%	28.6%	25.7%	18.2%	27.3%
D	16.7%	8.3%	22.9%	22.9%	36.4%	27.3%
F	8.3%	8.3%	11.4%	5.7%	9.1%	9.1%

In order not to distort the results, only data from face-to-face training is used – after the symbolic end of the pandemic. The control group consists of the students of the Informatics major (11 people), who in the 2nd semester are studying a mandatory course OOP in C++. The fact that the experimental group is heterogeneous is taken into account – for one major, the course is CS1, and for the other, CS2, but with an introductory C# course. Therefore, the first test is slightly different for the Software Engineering major. Language dependent questions are minimized as much as possible. The questions and tasks of the second test are the same for all and entirely related to the correct use of pointers, dynamic memory, and implementation of polymorphism, without specifying in what way.

Another circumstance that prevents accurate interpretation of the data is the small number of students in the control group. The Informatics major is currently the least desired of the three, and it has students ranked second and third preference, which is a demotivating factor.

Nevertheless, both the results of the control tests (Table I) and the direct observations of the students' activity in lectures, laboratory work, project work and homework, indicate that it makes sense to pay more attention to polymorphism in the OOP course. The experimental training conducted helps students to discover the exact relations between objects in the subject area more easily and correctly choose the operations that need to be implemented polymorphically. They are more adept at handling pointers and have a better understanding of the memory model. They recognize the situations in which they need to implement move semantics. They have no problem using lambda expressions in STL algorithms instead of function objects.

At the end of the course, a survey is conducted to determine students' satisfaction with the experiential learning and to specify what they found difficult. The answers are of interest. The question asked to the Software Engineering students was "Why did you choose C++?" (open question). 1/3 of them made such a choice, and 2/3 preferred PHP. Among the answers, it stands out that they read that it is suitable for Video game development, Embedded systems, Compilers and Enterprise software. The syntax of the methods implementing copy and move semantics – copy and move constructors and copy and move assignment operators – is indicated as the most difficult. Students find it difficult to navigate what is what just by the type of parameters. In second place are variable templates, and in third place – the many details of defining lambda expressions. When asked if what they learned about alternative ways of implementing polymorphism was useful, 73% answered yes.

IV. CONCLUSION

In this paper, the need for a more in-depth study of dynamic polymorphism is discussed. Problems associated with its implementation through inheritance and virtual functions are analyzed and a method of training is presented through a step-by-step project development with an alternative to inheritance design based on manual implementation of the functionality achieved with virtual tables. Specific guidelines for organizing training involving this topic are suggested.

As a recommendation to lecturers, it is useful to pay attention to the concept of polymorphism already in the introductory course. Although it is not dynamic there, it is good for students to recognize it early.

Regarding this implementation of polymorphism with type erasure, the most serious drawback is considered to be that, with the means currently available, it requires writing too much code – a code of high complexity. This is demotivating. Implementation with inheritance is much shorter and straight forward, even more so for students who have studied C#, where alternatives can easily be implemented – implementing interfaces can make it much easier to avoid inheritance. Such an implementation, of course, also has its drawbacks. However, if the proposals made to The C++ Standards Committee to introduce scalable reflection [8] and metaclasses [9] are implemented in the future, the code will be much shorter and clearer, because much of it will be automatically generated and will remain hidden from the client.

In studying the problem, it was found that not many researchers emphasize the weaknesses of inheritance as a tool to achieve polymorphism, and therefore there are not many proposals to overcome these weaknesses. The details in this area are mostly discussed at technical conferences rather than in scientific publications. This, on one hand, limited the possibilities of the research, and on the other, motivated the authors to tackle this problem.

Future work includes monitoring of the development of the language in this direction, although there has been some stagnation in the last 1-2 years. Also, in the future, it is planned to expand the experiment with formal statistical processing of accumulated empirical data.

References

[1] Milojković, N., Caracciolo, A., Lungu, M. F., Nierstrasz, O., Röthlisberger, D., Robbes, R., Polymorphism in the spotlight: studying its prevalence in Java and Smalltalk, ICPC '15: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, May 2015, pp. 186–195.

[2] Cardelli, L., Wegner, P., On understanding types, data abstraction, and polymorphism, ACM Computing Surveys, Volume 17, Issue 4 (December 1985), pp. 471-523.

[3] Stroustrup, Bj., 2012, Bjarne Stroustrup's C++ glossary, accessed 27 July 2022, https://www.stroustrup.com/glossary.html#Gpolymorphism.

[4] Brand, S., Dynamic polymorphism with metaclasses and code injection, talk at CppCon 2020, September 13-18, online, accessed 29 July 2022, https://www.youtube.com/watch?v=8c6BAQcYF_E.

[5] Tan, J., Guo, X., Zheng, W., Zhong, Ming., Case-based teaching using the Laboratory Animal System for learning C/C++ programming, Computers & Education, Volume 77, 2014, pp 39-49.

[6] Liberman, N., Beeri, C., & Kolikant, Y. B. D. (2011). Difficulties in learning inheritance and polymorphism. ACM Transactions on Computing Education, 11(1), pp 1–23, doi:10.1145/1921607.1921611.

[7] Milne, I., Rowe, G., Difficulties in learning and teaching programming – Views of Students and Tutors. Education and Information Technologies 7, pp. 55–66 (2002). https://doi.org/10.1023/A:1015362608943.

[8] Childers, W., Sutton, A., Vali, F., Vandevoorde, D. (2019), Scalable Reflection in C++, ISO/IEC C++ Standards Committee Papers, JTC1/SC22/WG21, Papers 2019, document P1240R1, mailing2019-10, www.open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1240r1.pdf.

[9] Sutter, H. (2019), Metaclass functions: Generative C++, ISO/IEC C++ Standards Committee Papers, JTC1/SC22/WG21 - Papers 2019, mailing2019-06, document P0707R4/2019-06-17, https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2019/p0707r4.pdf.