Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: «**ПОЛІМОРФІЗМ ТА ШАБЛОНИ. ПЕРЕВАГИ ТА НЕДОЛІКИ ОБОХ ПАРАДИГМ**»

Виконала студентка 4-го року навчання,

Освітньої програми “Інженерія Програмного Забезпечення”, 121

Калита Дарина Олександрівна

Керівник Бублик В.В.,
кандидат фіз.-мат. наук, доцент

Рецензент __Бучко О. А.____

Кваліфікаційна робота захищена
з оцінкою

Секретар ЕК _____

«____» ___ 2025 р.

Київ – 2025

ЗМІСТ Стор.

INTRODUCTION

Polymorphism and templates are two paradigms in programming that are both used to increase flexibility and reusability of code. However, they achieve that in different ways and both have their strong and weak points.

Polymorphism is commonly used in scenarios where runtime behavior customization is key, such as GUI toolkits (e.g., Qt), game engines, and enterprise software systems.

Templates, primarily found in languages like C++, enable generic programming. They allow code to be generated for specific types without requiring inheritance or virtual functions. This leads to increased efficiency.

While these paradigms serve different purposes and function differently, they are not mutually exclusive. On the contrary, their coexistence, if done correctly, leads to the detriments of both paradigms being minimized by their merits. There have been multiple discussions about the coexistence of templates and polymorphism, however, and while there is no doubt about the benefits of employing both paradigms at the same time, it is worth highlighting that this must be done with due care and diligence. Alexandrescu (2001) in “Modern C++ Design” notes , for example, that while templates and polymorphism can be powerful together, they also present design and debugging challenges that need to be carefully managed.

The purpose of this paper is to examine and analyze the advantages and the disadvantages of both polymorphism and templates in order to demonstrate ways in which the coexistence of these two paradigms can be beneficial. Ultimately, this analysis will offer insights into how developers can leverage these paradigms together to minimize their respectful downsides and create efficient and adaptable software.

The paper is composed of 2 parts:

The first part contains a brief history and general overview of both paradigms and an in-depth discussion of their pros and cons in general terms.

The second is dedicated to comparative analysis of the paradigms and ways in which they can interact, both positive and negative. Comparison tables are provided for visualisation and ease of comprehension and a functional example of a program is also presented, showcasing how the two paradigms can coexist and complement each other.

1. BACKGROUND

1.1. Polymorphism

The formal introduction of polymorphism is credited to Christopher Strachey in the 1960s. In his 1967 seminal paper “Fundamental Concepts in Programming Languages”, Strachey categorized polymorphism into two types: parametric and ad hoc types.

At first, inheritance and subclass polymorphism were not considered essential ingredients of OOP[4], but became a foundational feature of OOP with the rise of languages like Simula, the first programming language widely recognized as “object oriented”, and later C++ and Java. It has evolved over time to include both runtime (dynamic) and compile time (static) mechanisms, with different behaviour and different applications.

Polymorphism is now considered a cornerstone of OOP[5]. At its core, it allows objects of different types to be treated as objects of a common supertype. As a result, a program is able to use a certain function or method to process objects of different classes in ways that are specific to each class. Thus making it possible for one interface to display different behaviors.

Key characteristics of polymorphism include flexibility, as a single interface can handle multiple implementations; extensibility, reflected in how new types can be added with minimal changes to existing code; and adaptability, meaning the behavior adapts dynamically at runtime.[5]

The *Gang of Four* authors famously advised to “program to an interface, not an implementation”.[5] This principle is a through-thread of polymorphism and following it leads to clean and flexible code that is also easy to maintain.

Traditionally, polymorphism is categorized into two types[6]. One is runtime polymorphism, also called dynamic, which is achieved through inheritance and

virtual functions. In this case the method to be invoked is determined at runtime based on the actual object type. The second is compile-time polymorphism, or static polymorphism, contrastingly implemented through method overloading and operator overloading. The method is resolved during compilation.

Runtime (dynamic) polymorphism occurs, as its name suggests, at runtime and is achieved through method overriding. Meaning, a method that is already defined in a superclass or interface has specific implementations provided by subclasses. It is the object's runtime type instead of reference type that determines which method is invoked. In languages like C++ this behaviour is enabled by mechanisms like virtual tables (vtables) and in Java it functions via method dispatch.[7].

This type of polymorphism provides great flexibility and extensibility to the code it is being utilized in, especially in large systems[5]. This is made possible by binding to a specific method implementation commencing only during runtime. That has its trade-offs, however. For example, there is a slight runtime performance cost, which can become detrimental in larger systems[7].

Compile-time (static) polymorphism differs in the fact that it is resolved during compilation and is achieved through different means. Namely, through function overloading, operator redefinition, or templates (in languages like C++). In layman's terms, static polymorphism allows multiple methods or operators to have the same name but behave differently based on their parameters or context. Unlike runtime polymorphism, there's no overhead at execution because the decision about which function to invoke is made at compile time[8]. Furthermore, as Бублик demonstrates in his work on static double dispatch, compile-time polymorphism can be extended to replicate even traditionally dynamic behaviors, which enables even more efficient designs without the usual downsides.[2]

Beyond the common classification into runtime and compile-time forms, theoretical computer science and programming language theory further categorize polymorphism into more specific subtypes[9]:

- Ad-hoc polymorphism: Functions behave differently depending on the types of their arguments, typically via overloading or coercion.
- Subtype polymorphism: Enables treating a subclass as if it were an instance of its superclass, forming the basis of object-oriented inheritance.
- Parametric polymorphism: Generalizes code by making it type-agnostic, often implemented through templates in C++ or generics in Java.

However, normally it is not necessary to use such niche categorization.

In summary, polymorphism enables code to be reusable and easily extensible, lessens the need for duplication and provides great flexibility. Each of its forms serves a unique purpose. It can either be applied at runtime through inheritance and dynamic dispatch, at compile-time via method or operator overloading, or more abstractly through parametric and subtype polymorphism. It is essential to develop an understanding of these distinctions in order to be able to select the most appropriate approach when coding.

1.2 Templates

Templates enable generic programming. With it functions and classes are able to operate with any data type and don't require to be rewritten every time.

As a concept, generic programming was introduced by Alexander Stepanov in the late 1980s[10]. His work on algorithms that could operate on various types created the basis for what would become the Standard Template Library (STL) in C++. Formally templates were added to the C++ language with the C++98 standard[7], which introduced both function and class templates as first-class language features. STL was also adopted into the standard library at that time.

Templates are an example of compile-time polymorphism. Over time their capabilities were expanded with features such as template specialization, variadic templates (introduced in C++11)[8], and concepts (standardized in C++20)[11], which made generic programming even more expressive and type-safe.

At their core, templates serve as blueprints for a function or class that can operate with a generic type (e.g., T). The type or types are only specified when the template is instantiated. During compilation, separate implementations of the function or class are generated for each type[7].

Templates possess certain key features that they can be characterized by. Namely, one of them is type safety, meaning operations between incompatible types are prevented, another is efficiency, as code is generated at compile time, which prevents runtime type checking and, finally, code reusability, as multiple types are handled with the same template[7].

In C++, there are three types of templates. One is function templates, which enable the creation of functions that can work with any type. The second one is class templates, which allow the creation of class definitions that can handle different data types.[7] Since C++14 there have also been added variable templates, which

essentially are variables that function with the type specified when the variable is used[11]. No matter the type, templates are a compile-time construct.

1.3 Conclusions for Chapter 1

In this chapter the backgrounds of polymorphism and templates were outlined. Although they emerged from different historical contexts, both paradigms provide ways to write modular and reusable code.

Polymorphism, especially in its runtime form, is a fundamental aspect of OOP. It is a widely used tool for structuring large and constantly evolving codebases.

Templates, in turn, represent the core of generic programming in C++. Their capabilities have steadily evolved, with features such as variadic templates and concepts enhancing both expressiveness and reliability.

Ultimately, both paradigms serve distinct roles. Understanding the fundamental mechanics and history of these paradigms is essential for selecting the most appropriate one when solving real-world programming problems.

2. Advantages and disadvantages

2.1 Advantages of polymorphism

Understanding the strong sides of polymorphism is essential. Among them, flexibility stands out. As previously mentioned, in polymorphism, specifically runtime polymorphism, it is enabled through inheritance and polymorphic behavior.

Inheritance in itself, as a feature, has certain benefits. All derived classes adhere to a common interface that the base class defines, which ensures consistent behavior across different implementations[7]. Inherited methods and properties reduce duplication, because derived classes can share common behavior while also introducing their own features, specific to themselves.

Polymorphic behavior in turn allows objects to determine their behavior dynamically based on their actual type at runtime, even when accessed through a base class or interface reference. This enables the developer to write code that adapts to specific object types without needing explicit type checks or modifications. Functions operating on the base class automatically support new types.

Another advantage of polymorphism is its extensibility. New plugins can be integrated without modifying the core application, as long as they adhere to a common interface, which is in line with the Open/Closed Principle[12]. Media players or IDEs use polymorphism to support third-party plugins dynamically. Different types of UI components (buttons, sliders, text fields etc.) can share a common interface. Handling of different data types or processing steps also becomes way easier through polymorphic handlers.

Finally, polymorphism also greatly simplifies maintenance in large systems. It does so by promoting modularity and separation of concerns. As code which uses objects (clients) is separated from the specific implementations of those objects, individual components of a program can be updated, debugged, or replaced with

minimum effort and risk of errors. Also, individual components can be tested independently of their interactions with other parts of the system[13].

2.2 Disadvantages of polymorphism

While having plenty of benefits, polymorphism has its own detriments. One of them is performance overhead, which can be caused by dynamic dispatch[7][8]. In small-scale applications this overhead is usually negligible, but in performance-critical or real-time systems it can become a real issue[7]. Increased complexity in debugging and testing is another problem in and of itself[13].

Dynamic dispatch is a process which enables polymorphism by determining the correct method to execute based on the actual type of the object at runtime, not at compile time. This is achieved through virtual tables (vtables) and virtual pointers (vptrs)[7].

Vtables are tables maintained per class which contain pointers to the virtual functions of that class. Each derived class has its own vtable that overrides the base class's virtual functions where needed. Vptrs are hidden pointers stored in each object, pointing to the vtable of its actual class type. When a virtual function is called on an object, the vptr is used to look up the vtable for the object's type. After that the function pointer in the vtable is dereferenced to call the appropriate method.[7]

And that is wherein the issues lie. Instead of a direct function call (as in static polymorphism), the compiler generates an extra layer of indirection to find and invoke the correct function. This requires additional memory access and pointer dereferencing[8]. Accessing the vtable and resolving the function pointer can lead to cache misses, especially in performance-critical applications with large, complex inheritance hierarchies[7]. More memory is used, as each class requires a vtable, and each object includes a hidden vptr. There are inlining limitations too, since the actual method to call is not determined at compile time. This can impact optimization by the compiler[8].

In systems where performance is critical (for example, game engines, real-time systems, embedded devices), the overhead of dynamic dispatch can lead to noticeable performance degradation[7][2]. There can be CPU overhead, as the number of instructions increases due to vtable lookups and pointer dereferencing. The indirection can make it harder for the CPU to predict and optimize branching, which affects execution speed. Large systems with deep inheritance hierarchies have even more overhead due to more vtable lookups being performed even more frequently[8].

Another disadvantage of polymorphism is that it can make debugging and testing noticeably harder[13]. Methods are called indirectly, which makes the execution flow harder to predict and trace, and bugs in specific derived classes can remain hidden until those classes are used in a particular runtime scenario.

The difficulties arise because polymorphic behavior depends on runtime conditions and interactions between objects, which can obscure the flow of execution and make it harder to isolate issues[13].

Testing polymorphic code can have its own set of challenges, too. It is necessary to ensure full test coverage, missing one combination can result in imperfections and bugs remaining undetected. Aside from that, thorough regression testing is necessary, as changes in one class can inadvertently affect unrelated parts of the system due to shared base class behavior[13].

2.3 Advantages of templates

The compile-time nature of templates is what provides them the most advantages. One of those is the elimination of indirection. Unlike runtime polymorphism, templates do not rely on virtual tables. This means there is no need for dynamic dispatch and execution performance is better[14].

Another important advantage is that templates allow the compiler to inline small and frequently used functions easily, since the function bodies are available at compile time[8]. Furthermore, advanced compiler optimizations are enabled, for example loop unrolling and constant propagation. First is essentially a reduction of the overhead of loop control, achieved by executing multiple iterations of a loop within a single loop body. This minimizes branching and increases instruction-level parallelism. The second is a replacement of variables known to have constant values with those values during compilation which reduces computation and makes way for further optimizations like dead code elimination[1].

In contrast to polymorphism, which typically relies on an inheritance hierarchy to enable shared behavior among types, templates support reusability through compile-time type substitution. It allows generic code to operate across unrelated types, provided they satisfy the required interface. Avoiding a shared base class means developers are not forced to impose artificial relationships between otherwise unrelated types, which reduces unnecessary coupling and increases design flexibility[1][5][15].

2.4 Disadvantages of templates

While templates in C++ offer significant benefits through compile-time instantiation, their use is not without drawbacks. Among the most prominent issues is template expansion. In contrast to regular functions or classes that are compiled once and then simply reused, templates must be instantiated separately for every unique type. Each instantiation requires type checking, code generation, and optimization. As the number of instantiations grows, this can lead to substantially longer build times[8][14].

Another related drawback is so-called "code bloat". Because templates are instantiated for each specific type, the size of the resulting binary increases. This leads to longer load times and higher cache pressure at runtime. In applications where high performance is critical or systems with limited memory this can become a significant issue[14].

Templates are also known for producing compiler error messages that are hard to decipher. When an error occurs, like an unsupported operation being used or an incompatible type being passed, the compiler will display the full chain of instantiations. The message can span dozens of lines and the root cause of the problem may be deeply nested within technical details. This slows down debugging significantly, especially for less experienced developers[8].

The lack of built-in tools for template code inspection is another disadvantage of the paradigm, at least in the C++ programming language. As a workaround, advanced metaprogramming techniques such as SFINAE (Substitution Failure Is Not An Error), type_traits, or newer features like concepts are used. However, they increase code complexity and reduce readability[15]. Integration with systems that rely on dynamic type information or runtime behavior analysis is complicated by this too.

2.5 Chapter 2 conclusions

As this chapter has demonstrated, both paradigms bring their own set of strengths and limitations to the table. Each is most appropriate for different use cases.

Polymorphism promotes extensibility and modularity. These benefits prove to be particularly valuable in applications such as plugin architectures or UI frameworks. However, the advantages come at a cost. Specifically, runtime overhead, increased memory usage, and debugging and testing becoming more complex. These issues may be a reason to consider a different approach for systems where high performance is critical.

Templates, on the other hand, offer exceptional performance. They eliminate the need for dynamic dispatch and enable aggressive compiler optimizations. This makes them ideal for scenarios such as numerical computing or embedded development. Still, they have their own drawbacks. The compile times are longer, code bloat can happen much easier, and the debugging process is significantly harder, particularly for those unfamiliar with template metaprogramming techniques.

Ultimately, neither paradigm is universally superior. The choice between polymorphism and templates often depends on the specific software being developed. Systems which value runtime flexibility and require high extensibility may benefit more from polymorphism, while those which prioritize performance and type safety will gain from using templates. Many real-world C++ applications rely on a careful combination of both.

3. Comparative Analysis

Both polymorphism and templates are powerful paradigms that help achieve code generality and reuse. However, they differ significantly in benchmarks.

The following sections offer a side-by-side comparison.

Performance

Aspect	**Polymorphism**	**Templates**
Execution time	Slower: dynamic dispatch. Each virtual call involves a vtable lookup. T≈T_direct_call+O(1)vtable indirection[1][8]. May cause cache misses in tight loops.	Faster: no vtable lookup. Resolved at compile time. Often inlined, leading to T ≈ T_direct_call[1][2]. SIMD and loop unrolling more effective.
Memory use	Higher per object: each polymorphic object contains a hidden vptr (4–8 bytes)[8]. Vtable stored once per class.	Lower per object: no vptr. But total memory may increase due to multiple versions of the same code[8][2].
Compile time	Lower: base class and virtual function code is compiled once. T_compile≈T_base+T_derived.	Higher: templates are instantiated per unique type. T_compile≈N ×

		T_template[1][2][14]. Increased in complex templates (SFINAE, concepts).
Binary size	Smaller: shared implementation for all types via base class.	Larger: each instantiation results in separate code. Code bloat common if many types used.[1][14]

Explanation:

Templates offer better execution speed and lower memory usage per object due to compile-time resolution and the absence of virtual tables. However, this comes at the cost of longer compile times and larger binary sizes caused by code bloat from type-specific instantiations. Polymorphism, in contrast, provides more compact binaries and faster compilation through shared base implementations, but incurs runtime overhead due to dynamic dispatch.

Flexibility

Aspect	**Polymorphism**	**Templates**
Extensibility	High: new types are added by implementing a shared interface; existing code remains unchanged[1][8].	Moderate: adding new behaviors may require specializations or complex template constructs[1][2].

Runtime adaptability	Strong: decisions can be deferred until runtime, ideal for dynamic plugins and late binding[8][5].	Weak: behavior is fixed at compile time; no handling type variability at runtime[1][2].
Support for unrelated types	Limited: all polymorphic types derive from a common base, which enforces artificial coupling[8].	Strong: generic operations over unrelated types that satisfy required operations are allowed[1][2].
Code reuse model	Interface-based reuse via inheritance; promotes uniform design but enforces hierarchy[8].	Generic reuse across types via instantiation; supports broader application, especially in numeric and container code[1][2].

Explanation:

Templates offer better flexibility when working with unrelated types and enable high degrees of code reuse through generic constructs. However, polymorphism is more adaptable at runtime, making it better suited for systems that require dynamic behavior, such as plugin architectures.

Maintainability

Aspect	**Polymorphism**	**Templates**
Ease of	High: changing or extending	Lower: changes to templates can

modification	derived classes usually requires no change to base class or client code[8][5].	affect all instantiations and require full recompilation[1][2].
Testability	Easier: supports dependency injection, mocks, and interface testing at runtime[8][5].	Harder: testing requires generating test cases per instantiation; mocking is non-trivial[2][14]
Debugging experience	Better: mature tooling support and predictable runtime behavior[8][14].	Poorer: template errors can be hard to trace; error messages are often verbose and cryptic (esp. pre-C++20)[1][2].
Impact of change	Low: implementation details can change behind a stable interface[2].	High: template changes may ripple across all instantiations, risking subtle regressions[1][14].
Error isolation	Good: runtime bugs often localized to implementation or usage site[8].	Difficult: template instantiation errors can cascade and be hard to isolate, especially in heavily parameterized code[1][2].

Explanation:

Polymorphism generally leads to more maintainable code due to its stable interfaces, ease of modification, and strong tooling support. In contrast, templates—while efficient—can be harder to debug, test, and maintain due to complex error messages and the potential for widespread impact from small

changes. For long-term maintainability, polymorphism tends to be the more developer-friendly choice.

Readability and Complexity

Aspect	**Polymorphism**	**Templates**
Syntax clarity	Clearer: based on familiar OOP concepts (inheritance, interfaces, overrides)[8][5].	More cryptic: especially with metaprogramming, SFINAE, and decltype expressions[1][2].
Error messages	More readable: runtime and compile-time errors tend to be localized and short[8].	Often overwhelming: deep template errors can be dozens of lines long, with poor guidance (improved with concepts)[1][14].
Learning curve	Gentler: most programmers are introduced to OOP and inheritance early[5].	Steeper: requires knowledge of template syntax, type deduction, and possibly metaprogramming techniques[1][2].
Code conciseness	Moderate: explicit class hierarchies and virtual functions introduce boilerplate[8].	High: generic code can be concise but harder to understand, especially when heavily parameterized[2].
Tooling support	Mature: well-supported by debuggers, IDEs, and static	Varies: some tools struggle with template expansion, though support is

	analyzers[8][14]	improving in modern C++ compilers and IDEs[2][14].

Explanation:

Polymorphism is easier to read and understand. Templates on the other hand introduce syntactic and conceptual complexity, particularly when advanced techniques like metaprogramming are used. This makes template-heavy code less comprehensible, especially for beginners or during collaborative development.

In summary, both paradigms have their strengths and weaknesses. Relying on one exclusively leads to losses in performance, runtime flexibility or the maintainability of the system.

3.2 Conclusions of chapter 3

Polymorphism is invaluable in contexts where runtime adaptability and extensibility are critical. It enables developers to introduce new types or behaviors with minimal disruption to existing code, making it ideal for plugin architectures, UI frameworks, and other modular systems. However, the cost of dynamic dispatch and potential difficulties in debugging can become problematic, particularly in environments where resources are limited or high performance is paramount.

Templates, in contrast, shine when performance and compile-time guarantees are the main requirements. They allow developers to write highly optimized, type-specific code without runtime overhead and promote code reuse without enforcing artificial inheritance structures. This is especially beneficial in libraries such as Eigen or Boost, where generic operations must be both fast and type-safe. However, templates still have drawbacks. Code bloat is possible, the compile times are lengthened, and compiler errors are more complex, all of which hinder maintainability if not carefully managed.

The most effective strategy is to combine both approaches, leveraging their strengths while compensating for their respective weaknesses.

4. Coexistence of Polymorphism and Templates

4.1 Discussion of the topic

Many modern C++ designs benefit from combining both paradigms, leveraging each where it excels. Templates provide type-safe, reusable implementations optimized for performance[1], while polymorphism supports abstraction and extensibility through shared interfaces and late binding[8].

This coexistence is not only possible but advantageous, especially in large systems where both runtime adaptability and fine-grained compile-time optimization are required[1][2]. Templates eliminate the overhead of dynamic dispatch and allow compilers to generate specialized code for specific types[1]. Meanwhile, polymorphism facilitates the integration of new behavior at runtime, enabling the architecture to grow without recompiling existing components[8][5].

A practical way to bridge the gap between the paradigms is through the Curiously Recurring Template Pattern (CRTP). CRTP is a compile-time technique where a base class is templated on its derived class. This structure allows the base class to call methods in the derived class without relying on virtual functions, effectively simulating polymorphism at compile time[1]. CRTP enables behaviors such as static dispatch, interface enforcement, and even double dispatch without the runtime costs typically associated with inheritance. As Бублик[2] demonstrates in his work on static design patterns, this pattern is particularly powerful when performance is critical but a polymorphic structure is still desired.

In layered software designs, a division often emerges naturally. Templates are used for performance-critical internal logic, such as algorithms, containers, or numerical operations[1], while polymorphism is applied to high-level architectural elements, such as plugins, interfaces, and runtime strategies[8][5]. This separation allows teams to isolate optimizable components while preserving the maintainability and flexibility of the overall system.

Common real-world use cases for this hybrid approach include:

1. Generic containers with polymorphic elements
   A templated container (e.g., std::vector<T>) may store pointers or smart pointers to base class types. This enables runtime polymorphism within a generic, reusable container structure[1][8].

2. Strategy pattern with template-based strategies
   A high-level algorithm may be templated on a strategy interface, with runtime polymorphic implementations passed as arguments. This combines the performance of compile-time composition with the adaptability of runtime behavior[1][5].

3. Hybrid simulation or game engines
   In areas like game development or physical simulations, templates are often used for low-level math (e.g., vectors, matrices) or compile-time entity systems, while polymorphism structures the higher-level game object logic, event systems, or plugin architectures[1][5].

These patterns reflect a pragmatic balance between abstraction and specialization. Templates ensure correctness and efficiency in known, type-constrained contexts, while polymorphism allows systems to remain adaptable in the face of changing runtime requirements[1][8].

Considering everything that has been said, certain rules can be formulated in order to help developers decide when to use which paradigm or turn to their combination:

Templates are the best solution in cases where performance is key and types are known at compile time[1][2].

Polymorphism is most applicable when behavior varies at runtime and extensibility is required[8][5].

The coexistence of the paradigms is beneficial to systems which need to balance low-level optimization with high-level flexibility[1][5].

By combining compile-time and runtime techniques, developers are no longer forced to choose one paradigm at the expense of the other. Instead, they can build systems that are efficient, extensible, and tailored to the specific demands of each component.

4.2 Common examples and patterns

Modern C++ supports several design patterns that effectively integrate both templates and polymorphism. These patterns form the foundation of hybrid designs where both paradigms work together to maximize flexibility, performance, and maintainability[1][8].

One is the concept of policy-based design. By decoupling specific strategies (or "policies") from a class’s structure, this pattern enables static polymorphism while avoiding the runtime overhead of virtual functions[1]. It allows fine-grained control over class behavior, making it especially useful in configurable systems such as containers, logging systems, or memory allocators.

Mixins are another type of patterns which leverage the combination of polymorphism and templates. They are template-based classes used to inject additional functionality into a host class at compile time. Typically mixins are used to modularize features such as logging, serialization, or auditing. When layered onto polymorphic base classes, they enhance functionality without altering the virtual interface[1][2]. This approach maintains type safety and supports code reuse and provides a way to avoid multiple inheritance complexities.

Finally, there is the Curiously Recurring Template Pattern (CRTP). It is a form of static polymorphism in which a class template takes its derived class as a template argument. This enables the base class to access and invoke methods implemented in the derived class at compile time. CRTP eliminates the need for virtual functions, thereby removing dynamic dispatch overhead[1][2]. This pattern is suited especially well for performance-critical applications like mathematical libraries, component-based systems, and custom compile-time frameworks[2].

All these patterns demonstrate how templates and polymorphism can be layered to construct systems that are both efficient and adaptable.

To further illustrate how these paradigms complement each other, the following section presents a custom-developed project that leverages both runtime polymorphism and compile-time templating.

4.3 Practical example

This implementation highlights how design patterns like CRTP can be used to create systems that are both flexible and efficient, blending the adaptability of object-oriented design with the performance gains of generic programming.

```
template<class T>
struct S {
    /*Vector*/
    std::vector<T> v;
    /*Print out header*/
    S(std::initializer_list<T> l) : v(l) {
        std::cout << "constructed with a "
          << l.size() <<
          "-element list\n";
    }
    /*Append values to initializer_list<T>*/
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
};
```

The S<T> class demonstrates how templates support type-generic programming. By parameterizing the type T the same code can handle various data types such as int, double, or std::string, without needing separate implementations. This illustrates code reuse through compile-time type flexibility, which is a core strength of templates.

The constructor leverages std::initializer_list to support list-style initialization, and the append method shows how templated code can manipulate STL containers generically.

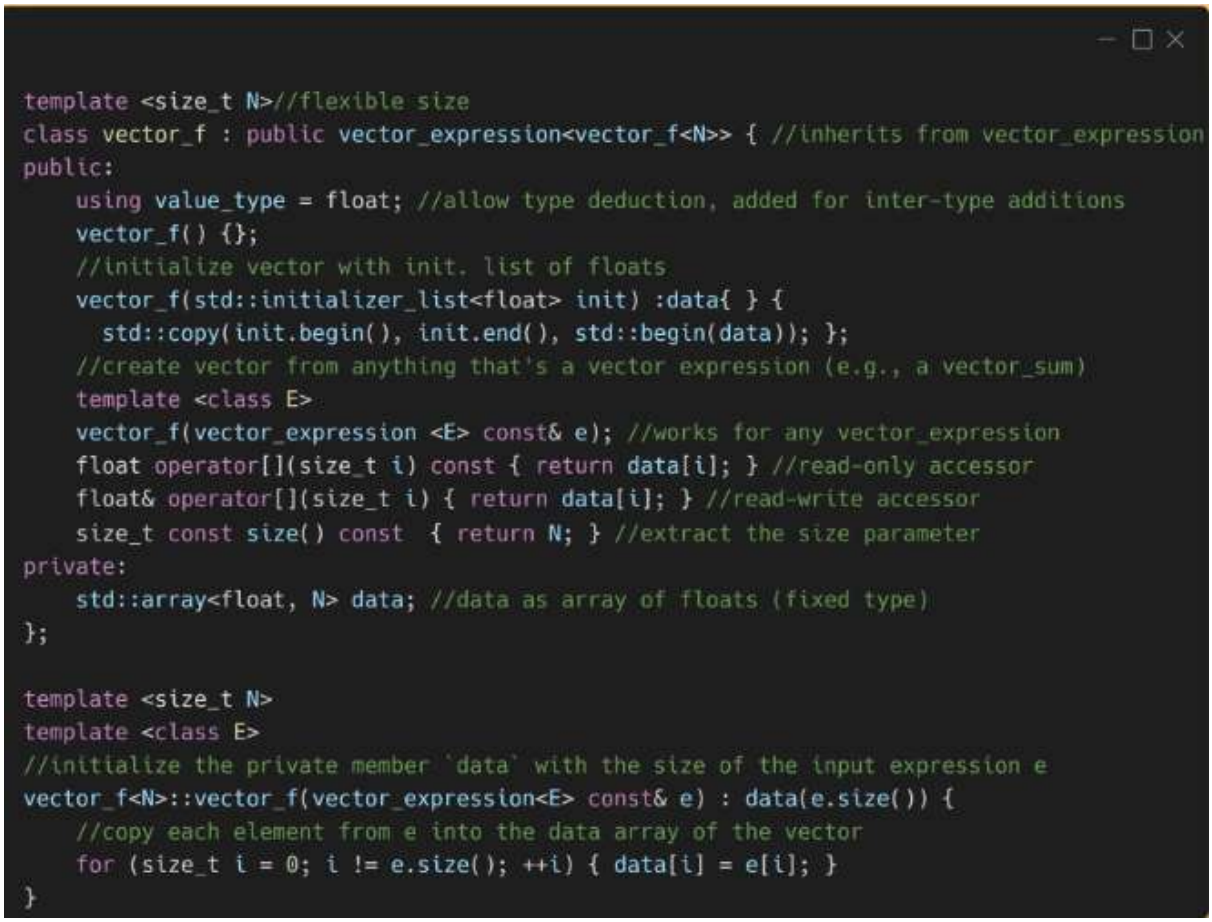
However, this implementation is purely compile-time and lacks any form of runtime polymorphism. There is no common base class or interface that enables treating multiple S<T> instances polymorphically, nor is there support for

overriding or extending behavior at runtime. As a result, adding new behaviors (such as type-specific logging or dynamic dispatch) would require modifying the class or writing a new version. A hybrid design could offer more flexibility by allowing runtime customization on top of the compile-time performance and type safety provided by templates.

First of all, a generic interface should be realized. The vector_expression<E> class template serves as this generic interface. It defines a consistent set of methods, namely operator[] and size(), that any derived expression type must implement. Using CRTP, vector_expression casts itself to the derived class to resolve the actual method implementations at compile time.

```
template <class E>
struct vector_expression {
    //Read-only access to elements of the expression
    float operator[](size_t i) const {
        //Cast to derived type and access the element
        return static_cast<E const&>(*this)[i];
    }
    //Vector size getter
    size_t size() const {
        //Cast to derived type and get size
        return static_cast<E const&>(*this).size();
    }
};
```

Unlike traditional polymorphic interfaces that rely on virtual functions and runtime dispatch, vector_expression enables static polymorphism. This provides many of the benefits of inheritance, like interface abstraction and substitution, without the cost of virtual tables. In essence, it allows different types of vector expressions to conform to a shared interface while retaining the full performance of template-based design. This is key in allowing the vector_f constructor to accept any expression conforming to the vector_expression<E> interface.



```
template <size_t N>//flexible size
class vector_f : public vector_expression<vector_f<N>> { //inherits from vector_expression
public:
    using value_type = float; //allow type deduction, added for inter-type additions
    vector_f() {};
    //initialize vector with init. list of floats
    vector_f(std::initializer_list<float> init) :data{ } {
      std::copy(init.begin(), init.end(), std::begin(data)); };
    //create vector from anything that's a vector expression (e.g., a vector_sum)
    template <class E>
    vector_f(vector_expression <E> const& e); //works for any vector_expression
    float operator[](size_t i) const { return data[i]; } //read-only accessor
    float& operator[](size_t i) { return data[i]; } //read-write accessor
    size_t const size() const  { return N; } //extract the size parameter
private:
    std::array<float, N> data; //data as array of floats (fixed type)
};

template <size_t N>
template <class E>
//initialize the private member `data` with the size of the input expression e
vector_f<N>::vector_f(vector_expression<E> const& e) : data(e.size()) {
    //copy each element from e into the data array of the vector
    for (size_t i = 0; i != e.size(); ++i) { data[i] = e[i]; }
}
```

The vector_f<N> class is a statically sized, float-based vector. Similar to the earlier S<T> example, this class is generic in structure. In this case, not by element type but by size, which is encoded as a template parameter. This approach enables firm control over memory layout and avoids dynamic allocations.

The class supports initialization from a list of floats, direct element access, and a read-write interface through overloaded subscript operators. Unlike traditional dynamic containers, vector_f stores its data in a fixed-size std::array<float, N>, ensuring predictable performance and allowing the compiler to optimize access patterns effectively. Importantly, it inherits from vector_expression<vector_f<N>>, using CRTP to enable participation in a larger expression system without runtime polymorphism. This inheritance introduces compile-time polymorphism, making vector_f interoperable with other expressions derived from vector_expression.

```cpp
template <class E1, class E2> class vector_sum :
                            public vector_expression<vector_sum<E1, E2>> {
public:
    // deduce common type
    using value_type = std::common_type_t<typename E1::value_type,
                                          typename E2::value_type>;
    /*Constructor takes two vector expressions as input*/
    vector_sum(const E1& u, const E2& v) :u(u), v(v) {};
    // compute sum of elements at index i, any types
    value_type operator[](size_t i) const {
        //convert elements
        return static_cast<value_type>(u[i]) + static_cast<value_type>(v[i]);
    }
    // returns the size of the sum (same as the size of the input vectors)
    size_t size() const { return v.size(); }
private:
    E1 const& u;//reference to the 1st vector_expr
    E2 const& v;//reference to the 2nd vector_expr
};
//Operator to sum 2 vectors deriving from vector_expression
template <typename E1, typename E2>
auto operator+(vector_expression<E1> const& u,
                            vector_expression<E2> const& v) {
    //Create and return vector_sum object representing the sum
    return vector_sum<E1, E2>(*static_cast<E1 const*>(&u),
                                    *static_cast<E2 const*>(&v));
}
```

The vector_sum class encapsulates the concept of lazy evaluation through expression templates. Instead of performing computation immediately, it acts as a lightweight proxy that stores references to two operand expressions. The logic for computing the element-wise sum is defined in its operator[], which computes each result on demand. This design eliminates the need for intermediate storage and allows complex expressions like a + b + c to be constructed and evaluated without allocating temporary vectors.

Crucially, vector_sum also inherits from vector_expression<vector_sum<E1, E2>>, making it composable and polymorphic at compile time. Because vector_sum is just another vector_expression, the consuming code does not need to distinguish

between vector_f, vector_sum, or other derived types and will word regardless of which one it is.

```cpp
template <typename T, size_t N>//templates
class my_vector : public vector_expression<my_vector<T, N>> {//polymorphism
private:
    std::array<T, N> data;
public:
    using value_type = T; //allow type deduction
    my_vector() : data{} {};//default constructor for vector<T,N>
    my_vector(std::initializer_list<T> init) {//constructor from an initializer list
        std::copy(init.begin(), init.end(), data.begin()); }
    //move constructor, ONLY if T is move-constructible
    my_vector(my_vector&& other) noexcept(std::is_move_constructible_v<T>) {
        //if T is_move_constructible, move data
        if constexpr (std::is_move_constructible_v<T>) {data = std::move(other.data);
        //if T is !is_move_constructible -> fallback to copy
        } else { data = other.data; }
    }
    //read-only accessor
    T operator[](size_t i) const { return data[i]; }
    //read-write accessor
    T& operator[](size_t i) { return data[i]; }
    //size getter
    constexpr size_t size() const { return N; }
    //get element type of vector
    void element_type() const { std::cout << typeid(T).name(); }
};
```

The my_vector<T, N> class combines template-based generality with compile-time polymorphism. Like S<T>, it supports storage and manipulation of various element types through template parameterization. However, my_vector generalizes this idea further by also supporting fixed-size optimization (N) and participating in a polymorphic framework via inheritance from vector_expression thanks to the Curiously Recurring Template Pattern (CRTP).

This dual-template structure enables high degrees of type safety, great performance, and helps with code reuse:

The T parameter allows the container to work with arbitrary types—float, double, user-defined types, etc., while maintaining compile-time correctness.

The size parameter N ensures predictable memory layout and optimization opportunities via std::array<T, N>.

The class provides both read-only and read-write access through overloaded operator[], and supports list-style initialization via std::initializer_list<T>.

Its move constructor selectively engages based on whether T is move-constructible, showcasing template metaprogramming with if constexpr and std::is_move_constructible.

Despite inheriting from a base interface, my_vector does not rely on runtime polymorphism, as there are no virtual functions. Instead, it leverages static polymorphism through CRTP, allowing the derived class to interact with polymorphic algorithms defined in vector_expression without the cost of virtual dispatch.

Unlike the basic S<T>, which provides type-generic behavior through a single template parameter but lacks extensibility and polymorphic structure, my_vector<T, N> builds on those strengths by introducing, first of all, two template parameters (T for type, N for size), which increase configurability and precision, and, second, compile-time polymorphic inheritance, which allows my_vector to plug into expression templates and generic vector frameworks.

Compared to vector_f<N>, which focuses on performance within a float-only numerical context, my_vector expands the design space by introducing full type generality, making it suitable for generic numerical computations across types. While vector_f is efficient and expression-template-aware, it is limited to a fixed element type. In contrast, my_vector offers the flexibility of S<T> and the CRTP-powered expression system of vector_f, making it a superior general-purpose container in terms of reusability and system integration.

Demonstrating the Coexistence of Templates and Polymorphism

The my_vector class embodies the coexistence of templates and polymorphism in a single, cohesive design. It inherits from a compile-time interface (vector_expression<E>), enabling it to integrate into a polymorphic system where different vector expressions can be combined, passed around, and evaluated without introducing dynamic overhead.

By abstracting behavior through a static interface while remaining fully type-agnostic and efficient, my_vector represents a culmination of the design goals explored throughout this thesis:

- Type generality through templates
- Structural extensibility through polymorphic composition
- Performance through static dispatch and inlining
- Flexibility through composition with other expression types

Together with vector_f and vector_sum, my_vector serves as a clear demonstration that templates and polymorphism are not competing paradigms but cooperative tools. When combined with care, they allow C++ developers to design systems that are modular, type-safe, and highly optimized—without sacrificing clarity or extensibility.

5. Conclusion

The goal of the thesis was to provide a clear understanding of how polymorphism and templates differ and how they can be used together with maximum effectiveness

The analysis showed that polymorphism is invaluable in systems that require flexibility and easy extensibility. It enables dynamic behavior through inheritance and interfaces and promotes modularity and clean architectural separation. However, these strengths come at the cost of potential runtime overhead and the increased complexity of debugging.

Templates, in contrast, enable highly efficient and type-safe code generation, which happens at compile time. They eliminate runtime dispatch, associated with polymorphism. Nonetheless, they also introduce issues such as longer compile times, code bloat, cryptic error messages, and they also lack built-in reflection mechanisms.

A central finding of this work is that these paradigms are not mutually exclusive. On the contrary, when used together, templates and polymorphism complement one another. Their combination offers a balance between flexibility and efficiency. This is especially evident in design patterns like CRTP, policy-based design, and mixins.

The thesis included code examples, particularly the use of CRTP and template-based vector operations, demonstrate how such hybrid designs can reduce the runtime costs typically associated with polymorphism while maintaining extensibility. The work of Бублик[2], which proposes statically dispatched alternatives for double dispatch, further supports this approach, showing how templates can help mitigate runtime limitations in polymorphic systems.

In conclusion, the most effective use of templates and polymorphism is not in isolation, but in a well thought out combination. By applying each paradigm where

it is most appropriate, developers can design systems that are not only efficient and scalable, but also adaptable and easy to maintain.

Джерела

1. Alexandrescu A. Modern C++ design: generic programming and design patterns applied. Addison-Wesley Professional, 2001. 352 p.
2. Boublik V. Towards creating a static design pattern for double dispatching model signatures. NaUKMA research papers. computer science. 2021. Vol. 4. P. 64–71. URL: https://doi.org/10.18523/2617-3808.2021.4.64-71 (date of access: 02.05.2025).
3. Strachey C. Fundamental Concepts in Programming Languages. *Higher-Order and symbolic computation*. 2000. Vol. 13, no. 1/2. P. 11–49. URL: https://doi.org/10.1023/a:1010000313106 (date of access: 06.05.2025).
4. Elliott E. The forgotten history of OOP. *Medium*. URL: https://medium.com/javascript-scene/the-forgotten-history-of-oop-88d71b9b2d9f (date of access: 06.05.2025).
5. Gamma E. Design patterns: elements of reusable object-oriented software. Reading, USA : Addison-Wesley, 1995. 395 p.
6. Sebesta R. W. Concepts of programming languages. Boston : Addison-Wesley, 2012.
7. Stroustrup B. The C++ programming language. Addison Wesley, 2013. 1368 p.
8. Meyers S. Effective modern C++. O'Reilly Media, Incorporated, 2014.
9. Cardelli L., Wegner P. On understanding types, data abstraction, and polymorphism. *ACM computing surveys*. 1985. Vol. 17, no. 4. P. 471–523. URL: https://doi.org/10.1145/6041.6042 (date of access: 22.05.2025).
10. Stepanov A. A. Elements of programming. Upper Saddle River, N.J : Addison-Wesley, 2009. 262 p.

11. Templates in C++ - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/templates-cpp/ (date of access: 25.05.2025).

12. Martin R. C. Agile software development, principles, patterns, and practices. 2nd ed. Prentice Hall, 2002. 529 p.

13. Booch G. Object-Oriented analysis and design with applications. San Jose : Addison-Wesley, 2007.

14. A. Alexandrescu, H. Sutter C++ coding standards: 101 rules, guidelines, and best practices / ed. by A. Andrei. Boston : Addison-Wesley, 2004. 240 p.

15. Stroustrup B. Design and evolution of C++. Pearson Education, Limited, 1994.

16. Lakos J. Large-scale C++ software design. Reading, Mass : Addison-Wesley Pub. Co., 1996. 846 p.

17. Niculescu V. Mixin based adaptation of design patterns. 16th international conference on evaluation of novel approaches to software engineering, Online Streaming, --- Select a Country ---, 26–27 April 2021. 2021. URL: https://doi.org/10.5220/0010444702610268 (date of access: 02.05.2025).

18. GeeksforGeeks. C++ Polymorphism - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/cpp-polymorphism/ (date of access: 02.05.2025).

19. Sahu A. DL infra series: c++ concepts – 2. Medium. URL: https://amritsahu-iitkgp.medium.com/dl-infra-series-c-concepts-2-6653a80cee62 (date of access: 02.05.2025).

20. Varlı C. C++ curiously recurring template pattern (CRTP). Medium. URL: https://cengizhanvarli.medium.com/c-curiously-recurring-template-pattern-crtp-737d5551fac6 (date of access: 02.05.2025).

21. Meyers S. More effective C++: 35 new ways to improve your programs and designs (addison-wesley professional computing series). Addison-Wesley Professional, 1995. 336 p.

22. Vandevoorde D., Josuttis N. M. C++ templates: the complete guide. Addison-Wesley Professional, 2002. 552 p.

23. Stroustrup B. Why C++ is not just an object-oriented programming language. Addendum to the proceedings of the 10th annual conference, Austin, Texas, United States, 15–19 October 1995. New York, New York, USA, 1995. URL: https://doi.org/10.1145/260094.260207 (date of access: 02.05.2025).

24. W3Schools.com. W3Schools Online Web Tutorials. URL: https://www.w3schools.com/cpp/cpp_polymorphism.asp (date of access: 02.05.2025).

25. C++ polymorphism. Programiz: Learn to Code for Free. URL: https://www.programiz.com/cpp-programming/polymorphism (date of access: 02.05.2025).

Код програми

init_list_vector.h

```
#pragma once
#include <initializer_list>
#include <iostream>
#include <vector>

template<class T>
struct S {
    /*Vector*/
    std::vector<T> v;
    /*Print out header*/
    S(std::initializer_list<T> l) : v(l) {
        std::cout << "constructed with a " << l.size() << "-element list\n";
    }
    /*Append values to initializer_list<T>*/
    void append(std::initializer_list<T> l) {
        v.insert(v.end(), l.begin(), l.end());
    }
};
```

part3.h

```
#pragma once
#include <cstddef>
#include <initializer_list>
#include <array>


//!!!crtp is used, compile-time polymorphism is used as well !!!

//Class for vector expressions
template <class E>
struct vector_expression {
    //Read-only access to elements of the expression
    float operator[](size_t i) const {
        //Cast to derived type and access the element
        return static_cast<E const&>(*this)[i];
    }
    //Vector size getter
    size_t size() const {
        //Cast to derived type and get size
```

```
        return static_cast<E const&>(*this).size();
    }
};

//Fixed-size vector class
template <size_t N>
class vector_f : public vector_expression<vector_f<N>> { //Inherits from
vector_expression
public:
    using value_type = float; //allow type deduction in vector_sum, added for inter-
type additions
    //Empty vector_f constructor
    vector_f() {};
    //Initialize vector with init. list of floats
    vector_f(std::initializer_list<float> init) :data{ } {  std::copy(init.begin(),
init.end(), std::begin(data)); };
    //Constructor to create vector from anything that is a vector expression (e.g., a
vector_sum)
    template <class E>
    vector_f(vector_expression <E> const& e); //To call, simply call, like a normal
vector. Difference: works for everything that's a vector_expression
    float operator[](size_t i) const { return data[i]; }// read-only accessor
    float& operator[](size_t i) { return data[i]; }// read-write accessor
    size_t const size() const  { return N; }// extract the size parameter
private:
    std::array<float, N> data; //Data of the vector_f (as vector_expression) as array
of floats
};

template <size_t N>
template <class E>
//Initialize the private member `data` with the size of the input expression e
vector_f<N>::vector_f(vector_expression<E> const& e) : data(e.size()) {
    //Copy each element from e into the data array of the vector
    for (size_t i = 0; i != e.size(); ++i) { data[i] = e[i]; }
}

//class for representing the sum of two vector expressions
template <class E1, class E2> class vector_sum : public
vector_expression<vector_sum<E1, E2>> {
public:
    using value_type = std::common_type_t<typename E1::value_type, typename
E2::value_type>; //deduce common type
    /*Constructor takes two vector expressions as input*/
```

```
    vector_sum(const E1& u, const E2& v) :u(u), v(v) {};
    //compute sum of elements at index i, any types
    value_type operator[](size_t i) const {
        return static_cast<value_type>(u[i]) + static_cast<value_type>(v[i]); //convert
elements
    }
    // returns the size of the sum (same as the size of the input vectors)
    size_t size() const { return v.size(); }
private:
    E1 const& u;//reference to the 1st vector_expr
    E2 const& v;//reference to the 2nd vector_expr
};
//Operator to sum 2 vectors deriving from vector_expression
template <typename E1, typename E2>
auto operator+(vector_expression<E1> const& u, vector_expression<E2> const&
v) {
    //Create and return vector_sum object representing the sum
    return vector_sum<E1, E2>(*static_cast<E1 const*>(&u), *static_cast<E2
const*>(&v));
}

my_vector.h
#pragma once
#include <array>
#include <initializer_list>
#include <memory>
#include <iostream>

//parametrization as required
template <typename T, size_t N>//templates
class my_vector : public vector_expression<my_vector<T, N>> {//polymorphism
private:
    std::array<T, N> data;
public:
    using value_type = T; //allow type deduction
    my_vector() : data{} {};//default constructor for vector<T,N>
    my_vector(std::initializer_list<T> init) {//constructor from an initializer list
        std::copy(init.begin(), init.end(), data.begin());
    }//move constructor, ONLY if T is move-constructible
    my_vector(my_vector&& other) noexcept(std::is_move_constructible_v<T>) {
        //if T is_move_constructible, move data
        if constexpr (std::is_move_constructible_v<T>) {data =
std::move(other.data);
        } else { data = other.data; }//if T is !is_move_constructible -> fallback to copy
```

```
    }
    //read-only accessor
    T operator[](size_t i) const { return data[i]; }
    //read-write accessor
    T& operator[](size_t i) { return data[i]; }
    //size getter
    constexpr size_t size() const { return N; }
    //get element type of vector
    void element_type() const { std::cout << typeid(T).name(); }
};
```

Main.cpp

```
#include <iostream>
#include "part3.h"
#include "init_list_vector.h"
#include "my_vector.h"


using namespace std;

void recursiveTemplateTest() {
    cout << "Testing S" << endl;
    S<int> s = { 1, 2, 3, 4, 5 }; // copy list-initialization

    cout << "Starting S: ";
    for (int i : s.v)
        cout << i << ' ';
    cout << endl;

    cout << "Let's append 6, 7, 8: ";

    s.append({ 6, 7, 8 }); // list-initialization in function call

    cout << "Full structure: ";
    for (int i : s.v)
        cout << i << ' ';
    cout << endl;
}


int main() {

    recursiveTemplateTest();
```

```
    cout << "\nTesting my_vector" << endl;
    my_vector<float, 3> v1;//zero-initialized
    my_vector<float, 3> v2{ 4.0f, 5.0f, 6.0f };//normal initialization

    //show off empty arr init & read-write accessor
    cout << "v1<float, " << v1.size() << ">" << endl;
    cout << "Filling up v1 with values..." << endl;
    v1[0] = 1.0f;
    v1[1] = 2.0f;
    v1[2] = 3.0f;

    cout << "v1<float, " << v1.size() << "> values: ";
    for (int i = 0; i < v1.size(); i++)
        cout << v1[i] << ' ';
    cout << endl;

    cout << "v2<float, " << v2.size() << "> values: ";
    for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << ' ';
    cout << endl;

    //show off vector_sum from PART_3 (for vector_expression)
    cout << "\n=SAME TYPE ADDITION=" << endl;
    auto  v3 = vector_sum(v1, v2);
    cout << "v3 = vector_sum(v1, v2), same type addition; size: " << v3.size() << ";
v3: ";
    for (int i = 0; i < v3.size(); i++)
        cout << v3[i] << ' ';
    cout << endl;

    //show off operator + from PART_3 (for vector_expression)
    auto v4 = v1 + v3;
    cout << "v4 = v1 + v3, same type addition; size: " << v4.size() << "; v4: ";
    for (int i = 0; i < v4.size(); i++)
        cout << v4[i] << ' ';
    cout << endl;

    cout << "\n====================\n";

    cout << "=DIFFERENT TYPE ADDITION=" << endl;
    cout << "testing different classes addition(same el.type)";
    my_vector <float, 3> v_mv = { 1.0f, 2.0f, 3.0f };//init list constructor shown
    vector_f<3> v_f = { 4.0f, 5.0f, 6.0f };
```

```
auto v_res = v_mv + v_f;

cout << "" << endl;

cout << "v_mv (" << typeid(v_mv).name() << "): ";
for (int i = 0; i < v_mv.size(); i++)
    cout << v_mv[i] << ' ';
cout << endl;

cout << "v_f (" << typeid(v_f).name() << "): ";
for (int i = 0; i < v_f.size(); i++)
    cout << v_f[i] << ' ';
cout << endl;

//print out result
cout << "v_mv + v_f: ";
for (int i = 0; i < v_res.size(); i++)
    cout << v_res[i] << ' ';
cout << endl;

cout << "testing different el. type addition (same classes)";
my_vector <double, 3> v_mv_d = { 0.1, 0.2, 0.3 };

cout << "\nv_mv_d (" << typeid(v_mv_d).name() << "): ";

for (int i = 0; i < v_mv_d.size(); i++)
    cout << v_mv_d[i] << ' ';
cout << endl;

cout << "v_mv (" << typeid(v_mv).name() << ");" << endl;

auto v_res2 = v_mv_d + v_mv;

cout << "v_mv_d + v_mv: ";
for (int i = 0; i < v_res2.size(); i++)
    cout << v_res2[i] << ' ';
cout << endl;

//different el. type and classes

auto v_res3 = v_mv_d + v_f;

cout << "(different type and class) v_mv_d + v_f: ";
for (int i = 0; i < v_res3.size(); i++)
```

```
        cout << v_res3[i] << ' ';
    cout << endl;


        return 0;
}
```